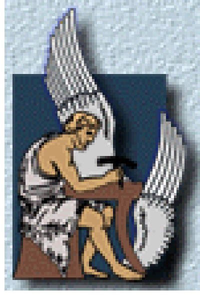


ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ ΚΑΙ ΥΛΙΚΟΥ

Διπλωματική Εργασία

«Υλοποίηση Αλγορίθμων TSP, Minisat και Genetic σε Επεξεργαστή και Συνεπεξεργαστή με Αναδιατασσόμενη Λογική»

ΠΛΗΚΑΔΙΤΗ ΑΓΟΡΙΤΣΑ

Εξεταστική επιτροπή:

Παπαευσταθίου Ιωάννης	Δόλλας Απόστολος	Πνευματικάτος Διονύσιος
Επίκουρος Καθηγητής	Καθηγητής	Αναπληρωτής Καθηγητής
(Επιβλέπων)		

Χανιά, Ιούνιος 2008

ΕΥΧΑΡΙΣΤΙΕΣ

Ολοκληρώνοντας τη διπλωματική αυτή εργασία, θα ήθελα να ευχαριστήσω όλους όσους μου συμπαραστάθηκαν κατά τη διάρκεια της φοίτησής μου στο Πολυτεχνείο Κρήτης.

Παράλληλα αισθάνομαι την ανάγκη να ευχαριστήσω τον καθηγητή μου κ. Ιωάννη Παπαευσταθίου, για την επίβλεψη καθώς και την σημαντική καθοδήγησή του κατά την διάρκεια της διπλωματικής μου εργασίας.

Ακόμη θα ήθελα να ευχαριστήσω τους καθηγητές κ. Απόστολο Δόλλα και κ. Διονύσιο Πνευματικάτο για τον χρόνο που αφιέρωσαν να διαβάσουν το κείμενο αυτό και για τις παρατηρήσεις τους.

Επίσης, τον Διδακτορικό φοιτητή του Εργαστηρίου Μικροεπεξεργαστών και Υλικού, Γρηγόρη Χρυσό, για την πολύτιμη βοήθεια, καθοδήγηση και στήριξη την οποία μου παρείχε καθ' όλη τη διάρκεια της εκπόνησης της εργασίας.

Επιπλεόν, θα ήθελα να ευχαριστήσω τον Μανώλη Κολοκοτρώνη, τελειόφοιτο του Πολυτεχνείου Κρήτης, για τις συμβουλές και τις ιδέες του σε δύσκολα σημεία της εργασίας.

Τέλος, νιώθω την ανάγκη να ευχαριστήσω την οικογένειά μου αλλά και τους φίλους μου για την στήριξη που μου παρείχαν όλα αυτά τα χρόνια.

Πληκαδίτη Αγορίτσα

Χανιά, Ιούνιος 2008

ΠΕΡΙΛΗΨΗ

Η σχέση βέλτιστου αποτελέσματος – χρόνου έχει απασχολήσει και συνεχίζει να απασχολεί τους σχεδιαστές ενσωματωμένων κυκλωμάτων και όλους όσους ασχολούνται με την επιστήμη υπολογιστών.

Νέες τεχνολογίες αναπτύσσονται συνέχεια με σκοπό να βελτιστοποιήσουν τα προγράμματα και να τα κάνουν να τρέχουν σε όσο το δυνατό λιγότερο χρόνο και με λιγότερους κύκλους ρολογιού. Μια σχετικά πρόσφατη προσπάθεια στον τομέα αυτό έγινε από την εταιρεία Stretch.

Πολλά είναι τα προβλήματα στα οποία η σχέση βέλτιστου αποτελέσματος – χρόνου είναι αμφιλεγόμενη και τρία από αυτά είναι το πρόβλημα του πλανόδιου πωλητή, το πρόβλημα της ικανοποιησιμότητας μιας λογικής πρότασης καθώς και οι γενετικοί αλγόριθμοι.

Στην παρούσα εργασία προσπαθούμε να δούμε σε τι ποσοστό είναι δυνατό να βελτιστοποιήσουμε τα παραπάνω τρία προβλήματα με τα εργαλεία της Stretch.

Λέξεις κλειδιά: *TSP, MINISAT, Γενετικός Αλγόριθμος, Stretch, Βελτιστοποίηση, Συνολικοί κύκλοι ρολογιού, Data Cache Misses*

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 1 : Εισαγωγή..7

1.1 Αντικείμενο της διπλωματικής...7

1.2 Δομή της εργασίας...8

ΚΕΦΑΛΑΙΟ 2: Οι Αλγόριθμοι..10

2.1 Το Πρόβλημα του Πλανόδιου Πωλητή
(Travelling Salesman Problem, TSP)...10

2.1.1 Εισαγωγή και Ιστορικά Στοιχεία..10

2.1.2 Ευριστικοί Αλγόριθμοι Κατασκευής της Διαδρομής.................14

2.1.2.1 Nearest Neighbor Algorithm (NN)..............................14

2.1.2.2 Greedy Algorithm...15

2.1.2.3 Nearest Insertion Algorithm..17

2.1.3 Αλγόριθμοι που χρησιμοποιήθηκαν...18

2.1.4 Εφαρμογές του TSP...21

2.2 Το πρόβλημα ελέγχου της ικανοποιησιμότητας
μιας λογικής πρότασης (Boolean Satisfiability Problem, SAT)...............22

2.2.1 Εισαγωγή...22

2.2.2 Προδιαγραφές Προβλήματος... 23

2.2.3 MINISAT...23

2.2.3.1 Εισαγωγή...23

2.2.3.2 Περίληψη του SAT-solver..24

2.2.3.2.1 Διάδοση (propagation)..26

2.2.3.2.2 Learning (Εκμάθηση)...27

2.2.3.2.3 Search (Εύρεση)...28

2.2.3.2.4 Activity (Δραστηριότητα)...30

2.2.3.2.5 Constraint removal (Απομάκρυνση περιορισμών)....30

2.2.3.2.6 Top-level solver (επίλυση
στο επίπεδο απόφασης 0).......................................31

2.3 Γενετικός Αλγόριθμος (Genetic Algorithm, GA)....................................32

2.3.1 Εισαγωγή και Ιστορικά Στοιχεία..32

2.3.2 Μεθοδολογία..33

2.3.2.1 Επιλογή (Selection)...36

2.3.2.2 Αναπαραγωγή (Mutation, Crossover)..........................36

2.3.2.3 Τερματισμός...37

2.3.2.4 Ψευδοκώδικας...38

2.3.3 Πλεονεκτήματα και μειονεκτήματα γενετικών αλγορίθμων και εφαρμογές τους...............................38

2.3.4 Επίλυση του TSP με γενετικό αλγόριθμο.................................39

ΚΕΦΑΛΑΙΟ 3: Η Τεχνολογία Stretch και Σχετικές Μέθοδοι Βελτιστοποίησης...43

3.1 Στόχος της Stretch...43

3.2 Η αρχιτεκτονική της Stretch..44

3.3 Το περιβάλλον ανάπτυξης της Stretch...46

3.4 Βελτιστοποίηση με τους επεξεργαστές της Stretch.................................48

3.4.1 Εύρεση compute intensive τμημάτων κώδικα............................48

3.4.2 Βελτιστοποίηση κώδικα...49

3.4.2.1 Χρήση “προσαρμοζόμενων” εντολών..........................49

3.4.2.2 Εκτέλεση Υπολογισμών σε Παραλληλισμό................51

3.4.2.3 Manual Loop Unrolling...51

3.4.2.4 Χρήση μνήμης υψηλών επιδόσεων.............................52

3.4.2.5 Περαιτέρω βελτιστοποιήσεις.......................................52

ΚΕΦΑΛΑΙΟ 4: Βελτιστοποίηση αλγορίθμων TSP και MINISAT....54

4.1 Υλοποίηση TSP...54

4.2 TSP και Stretch...56

4.3 Συμπεράσματα..59

ΚΕΦΑΛΑΙΟ 5: Βελτιστοποίηση του MINISAT................................61

5.1 Υλοποίηση MINISAT...61
5.2 MINISAT και Stretch...63
5.3 Συμπεράσματα..66

ΚΕΦΑΛΑΙΟ 6: Βελτιστοποίηση Γενετικού Αλγορίθμου...................67

6.1 Υλοποίηση Γενετικού Αλγορίθμου...67
6.2 Γενετικός Αλγόριθμος και Stretch……………………….…………..69
6.3 Συμπεράσματα..74

ΚΕΦΑΛΑΙΟ 7: Συμπεράσματα και Μελλοντική Εργασία................75

7.1 Συμπεράσματα..75
7.2 Μελλοντική Εργασία..77

Παράρτημα..79

Βιβλιογραφία...89

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

1.1 Αντικείμενο της διπλωματικής

Η ραγδαία ανάπτυξη των υπολογιστών και η είσοδός τους στην καθημερινή μας ζωή αλλά και σε όλα τα επιστημονικά πεδία, έχει ως συνέπεια την προσπάθεια επίλυσης διαφόρων προβλήματων με τη βοήθειά τους. Ταυτόχρονα η ζωή μας με τους γρηγόρους ρυθμούς της απαιτεί σχεδόν σε όλους τους τομείς της τη βέλτιστη λύση σε όσο το δυνατό λιγότερο χρόνο. Συνδυάζοντας τους δύο αυτούς παράγοντες καταλαβαίνουμε ότι ο στόχος και η πρόκληση για όλους όσους ασχολούνται με την επίλυση προβλημάτων με χρήση ηλεκτρονικών υπολογιστών είναι να μειώσουν τη σχέση βέλτιστου αποτελέσματος – χρόνου. Για το σκοπό αυτό αναπτύχθηκαν πολλές εφαρμογές (τόσο για υλικό όσο και για λογισμικό), οι οποίες χρησιμοποιήθηκαν με στόχο τη δημιουργία προγραμμάτων που δίνουν λύσεις στα προβλήματα αυτά με όσο το δυνατόν μεγαλύτερη ακρίβεια στο συντομότερο χρόνο.

Υπάρχουν αρκετά προβλήματα στα οποία μας ενδιαφέρει να μελετήσουμε τη σχέση βέλτιστου αποτελέσματος – χρόνου και αυτά είναι που έχουν τραβήξει τη μεγαλύτερη προσοχή των ανθρώπων που ασχολούνται με την επιστήμη των υπολογιστών. Δύο από τα προβλήματα

αυτά είναι το πρόβλημα του πλανόδιου πωλητή (TSP) και το πρόβλημα της ικανοποιησιμότητας μιας λογικής πρότασης (SAT).

Παράλληλα, μεγάλο ενδιαφέρον σημειώνεται στην ανάπτυξη συστημάτων επίλυσης προβλημάτων βασισμένων στις αρχές της φυσικής εξέλιξης, χαρακτηριστικό παράδειγμα των οποίων είναι οι γενετικοί αλγόριθμοι (GA). Τα αποτελέσματα των γενετικών αλγορίθμων αποτελούν ακόμα ένα πρόβλημα στο οποίο μας ενδιαφέρει να έχουμε το βέλτιστο αποτέλεσμα στο λιγότερο χρόνο.

Σε πολλές περιπτώσεις είναι επιθυμητή η βελτιστοποίηση αυτή των προβλημάτων να γίνεται σε υλικό αντί για λογισμικό. Μια εφαρμογή που έχει αναπτυχθεί τα τελευταία χρόνια και έχει ως σκοπό να βοηθήσει τους μηχανικούς ενσωματωμένων συστημάτων στη δημιουργία γρήγορων επεξεργαστών που θα επιταχύνουν ακόμα και τα πιο απαιτητικά προβλήματα είναι η Stretch. Στην παρούσα εργασία μελετούμε κατά πόσο μπορούν να βελτιστοποιηθούν τα τρία προβλήματα που αναφέραμε προηγουμένως (TSP, SAT, GA) με χρήση των εργαλείων της Stretch.

1.2 Δομή της εργασίας

Ο τόμος της διπλωματικής εργασίας αποτελείται από 5 κεφάλαια. Στα κεφάλαια αυτά γίνεται αναλυτική περιγραφή της διαδικασίας η οποία ακολουθήθηκε κατά την εκπόνηση της εργασίας. Συγκεκριμένα, περιγράφονται έννοιες σχετικές με το θέμα και επιπλέον, παρουσιάζεται το εργαλείο το οποίο χρησιμοποιήθηκε, τα αποτελέσματά μας και ο τρόπος με τον οποίο φτάσαμε σε αυτά.

Πιο αναλυτικά, το 1ο κεφάλαιο είναι εισαγωγικό και περιγράφει σε γενικές γραμμές το αντικείμενο της εργασίας.

Στο κεφάλαιο 2 γίνεται μια ανάλυση των αλγορίθμων που μας απασχόλησαν, των μεθόδων που μπορούν να χρησιμοποιηθούν για την επίλυσή τους και μια πρώτη αναφορά στην υλοποίηση που επιλέξαμε.

Στο κεφάλαιο 3 δίνονται κάποιες πληροφορίες για το εργαλείο το οποίο χρησιμοποιήσαμε στην παρούσα εργασία.

Στα κεφάλαια 4 και 5 παρουσιάζεται αναλυτικά η προσπάθειά μας να βελτιστοποιήσουμε τους αλγορίθμους και τα αποτέλεσματά μας συνολικά αλλά και μετά από κάθε προσπάθεια βελτιστοποίησης.

ΚΕΦΑΛΑΙΟ 2

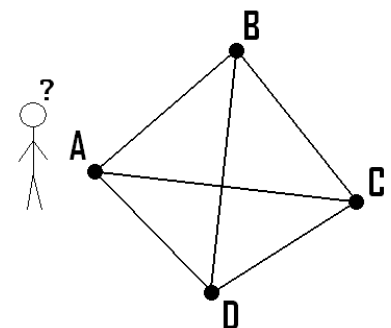
Οι Αλγόριθμοι

Στο κεφάλαιο αυτό θα γίνει μια πρώτη γνωριμία με τους αλγορίθμους με τους οποίους εργαστήκαμε δίνοντας τα βασικά χαρακτηριστικά τους, τις αρχές λειτουργίας τους, κάποιες εφαρμογές τους και ψευδοκώδικα όπου αυτό είναι δυνατό.

2.1 Το Πρόβλημα του Πλανόδιου Πωλητή (Travelling Salesman Problem, TSP)

2.1.1 Εισαγωγή και Ιστορικά Στοιχεία

Το πρόβλημα του πλανόδιου πωλητή (TSP) είναι ένα πρόβλημα στο οποίο μας δίνεται μια συλλογή από σημεία που αντιστοιχούν σε πόλεις και οι μεταξύ τους αποστάσεις (κόστη) και μας ζητείται να υπολογίσουμε τη διαδρομή με το μικρότερο κόστος περνώντας από όλες τις πόλεις μία φορά και επιστρέφοντας στην αρχική.



Σχήμα 2.1

Από το 1800 ακόμα ο Ιρλανδός μαθηματικός Sir William Rowan Hamilton και ο βρετανός μαθηματικός Thomas Kirkman ασχολήθηκαν με προβλήματα που σχετίζονται με το TSP και μια πρώτη προσέγγιση της δουλειάς τους μπορεί να βρεθεί στο *Graph Theory 1736-1936.* Η γενική μορφή του TSP όμως μελετήθηκε και πρωτοεμφανίστηκε από μαθηματικούς το 1930 στη Βιέννη και το Χάρβαρντ, και συγκεκριμένα από τον Karl Menger. Αργότερα το πρόβλημα μελέτησαν οι Hassler Whitney και Merrill M. Flood στο Πρίνστον. Μια λεπτομερή μελέτη των συνδέσεων μεταξύ των Menger και Whitney καθώς επίσης και η εξέλιξη της μελέτης του TSP μπορούν να βρεθούν στο σύγγραμα του Alexander Schrijver's "*On the history of combinatorial optimization (till 1960)*" που δημοσιεύτηκε το 2005.

Μέχρι και σήμερα το TSP παραμένει ένα από τα πιο δημοφιλή μαθηματικά προβλήματα και παρόλο που έχει προσελκύσει μεγάλο ποσοστό μελετητών δεν έχει βρεθεί ακόμα λύση για τη γενική περίπτωση. Το πρόβλημα προκύπτει όταν ο αριθμός των πόλεων (N) είναι μεγάλος γιατί κανένας υπολογιστής δε μπορεί να υπολογίσει την βέλτιστη διαδρομή σε εύλογο χρονικό διάστημα. Αυτό οφείλεται στο μεγάλο αριθμό των πιθανών υπολογισμών που είναι *O((N-1)!).* Τέτοιου είδους προβλήματα καλούνται NP-hard (όπου NP σημαίνει "non-deterministic polynomial") και όλα αυτά τα προβλήματα έχει αποδειχτεί ότι είναι ισοδύναμα, δηλαδή αν βρεθεί λύση για ένα από αυτά θα ακολουθήσουν οι λύσεις και για τα υπόλοιπα. Οι κύριες μέθοδοι επίλυσης των προβλημάτων αυτών είναι:

- Αλγόριθμοι που βρίσκουν την ακριβή λύση και δουλεύουν "γρήγορα" για προβλήματα σχετικά μικρού μεγέθους.

- Ευριστικοί ή αλγόριθμοι που βρίσκουν λύσεις κοντά στη βέλτιστη (sub-optimal), δηλαδή αλγόριθμοι που δίνουν αποδεδειγμένα καλές λύσεις αλλά που δεν αποδεικνύεται πως είναι βέλτιστες [3]. Για την περίπτωση των ευριστικών πολλοί αλγόριθμοι έχουν προταθεί να προσφέρουν "γρήγορες" και ταυτόχρονα "καλές" λύσεις. Τα τελευταία χρόνια μάλιστα έχουν προταθεί λύσεις, για μεγάλα προβλήματα (10000 – 15000 πόλεις ή περισσότερες), οι οποίες τρέχουν σε λογικά χρονικά πλαίσια και απέχουν ελάχιστα από τη βέλτιστη λύση [3,4].

Αν προχωρήσουμε ακόμα περισσότερο μπορούμε να κατατάξουμε τους ευριστικούς αλγορίθμους, που χρησιμοποιούνται για την επίλυση του TSP, σε δύο κατηγορίες:

- Αλγόριθμοι που χρησιμοποιούνται για την κατασκευή της διαδρομής, οι οποίοι εκτελούν μια ακολουθία από πράξεις μέχρι να έχουμε μια έγκυρη διαδρομή και όταν την αποκτήσουμε σταματούν και επιστρέφουν τη διαδρομή αυτή.
- Αλγόριθμοι που χρησιμοποιούνται για τη βελτίωση της διαδρομής, που έχει κατασκευαστεί με κάποιον από τους αλγορίθμους της προηγούμενης κατηγορίας, οι οποίοι ξεκινούν από μία έγκυρη διαδρομή και επαναληπτικά βελτιώνουν το κόστος της μέχρι να ικανοποιήσουν κάποια προκαθορισμένη συνθήκη οπότε και τερματίζουν.

Με βάση αποτελέσματα που εμφανίζονται στο [1], οι αλγόριθμοι κατασκεύης της διαδρομής είναι κατά 8% χειρότεροι από τις μεθόδους βελτιστοποίησης της διαδρομής [5].

Τέλος πρέπει να αναφέρουμε ότι ο σύνηθες τρόπος για την εισαγωγή των πόλεων και των αποστάσεών τους είναι η βιβλιοθήκη TSPLIB, η μορφή της οποίας φαίνεται στο *Σχήμα 2.2*.

Από πόλη σε πόλη:	Αθήνα	Αλεξανδ-ρούπολη	Άμφισ-σα	Άρτα	Βέροια	Βόλος	Γρεβενά	Δράμα	Έδεσ-σα	Ηγου-μενίτσα	Θεσσα-λονίκη	Ιωάν-νινα	Καβάλα	Καλα-μάτα	Καρδί-τσα	Καρπε-νήσι	Καστο-ριά	Κατε-ρίνη	Κιλκίς
Αθήνα		854	200	362	520	324	512	684	555	479	515	438	680	284	305	293	593	446	565
Αλεξανδρούπολη	854		712	779	414	556	529	212	427	792	346	702	177	1055	550	650	535	411	369
Άμφισσα	200	712		273	347	189	282	526	428	393	373	341	538	331	164	151	374	303	424
Άρτα	362	779	273		348	347	254	577	394	147	425	76	589	380	248	200	345	369	468
Βέροια	520	414	347	348		224	114	230	50	368	76	292	243	721	255	355	150	75	112
Βόλος	324	556	189	347	224		213	372	273	367	216	274	383	526	124	192	292	149	267
Γρεβενά	512	529	282	254	114	213		362	143	269	192	178	355	620	115	288	92	186	233
Δράμα	684	212	526	577	230	372	362		264	644	170	540	37	800	427	580	396	271	220
Έδεσσα	555	427	428	394	50	273	143	264		433	94	339	259	671	283	443	132	109	120
Ηγουμενίτσα	479	792	393	147	368	367	269	644	433		474	372	639	574	280	370	367	401	510
Θεσσαλονίκη	515	346	373	425	76	216	192	170	94	474		370	165	715	257	410	226	101	50
Ιωάννινα	438	702	341	76	292	274	178	540	339	372	370		535	457	176	266	263	297	406
Καβάλα	680	177	538	589	243	383	355	37	259	639	165	535		878	422	575	391	266	215
Καλαμάτα	284	1055	331	380	721	526	620	800	671	574	715	457	878		504	411	690	646	745
Καρδίτσα	305	550	164	248	255	124	115	427	283	280	257	176	422	504		170	208	148	268
Καρπενήσι	293	650	151	200	355	192	288	580	443	370	410	266	575	411	170		379	308	428
Καστοριά	593	535	374	345	150	292	92	396	132	367	226	263	391	690	208	379		227	224
Κατερίνη	446	411	303	369	75	149	186	271	109	401	101	297	266	646	148	308	227		120
Κιλκίς	565	369	424	468	112	267	233	220	120	510	50	406	215	745	268	428	224	120	

Σχήμα 2.2

2.1.2 Ευριστικοί Αλγόριθμοι Κατασκευής της Διαδρομής

Γενικά το πρόβλημα του πλανόδιου πωλητή είναι ένα πρόβλημα βελτιστοποίησης το οποίο μπορεί να προσεγγιστεί με διάφορους αλγορίθμους. Στην ενότητα αυτή θεωρήσαμε σωστό να παρουσιάσουμε ορισμένους σημαντικούς αλγορίθμους κατασκευής της αρχικής διαδρομής.

2.1.2.1 Nearest Neighbor Algorithm (NN)

Ο αλγόριθμος του πλησιέστερου γείτονα (Nearest Neighbor Algorithm, NN) είναι ένας από τους πρώτους αλγορίθμους που χρησιμοποιήθηκαν για την επίλυση του TSP και μας παρέχει γρήγορα μια σύντομη διαδρομή αλλά συνήθως όχι τη βέλτιστη. Ο αλγόριθμος αυτός “μιμείται” τον πωλητή ο οποίος πάντα ταξιδεύει στην αμέσως κοντινότερη πόλη, που δεν έχει επισκεφτεί, από αυτή που βρίσκεται.

Τα ακριβή βήματα του NN είναι:

1. Επιλέγουμε ένα τυχαίο σημείο σαν το τρέχον σημείο.
2. Βρίσκουμε τη διαδρομή με το μικρότερο κόστος που ενώνει το τρέχον σημείο και ένα σημείο που δεν έχουμε επισκεφτεί V.
3. Ορίζουμε ως τρέχον σημείο το V.
4. Σημειώνουμε ότι το σημείο V το έχουμε επισκεφτεί.
5. Αν έχουμε επισκεφτεί όλα τα σημεία τερματίζουμε.
6. Διαφορετικά επιστρέφουμε στο βήμα 2.

Η ακολουθία των σημείων που έχουμε επισκεφτεί είναι και η έξοδος του αλγορίθμου.

Όπως φαίνεται ο NN είναι εύκολος στην υλοποίηση και εκτελείται γρήγορα, αλλά μερικές φορές δεν βρίσκει διαδρομές που είναι σύντομες, και μπορεί εύκολα να τις αντιληφθεί κανείς με μια ματιά, εξαιτίας της "άπληστης" φύσης του. Γενικά, αν τα τελευταία στάδια της διαδρομής μπορούν να συγκριθούν με τα αρχικά, τότε η διαδρομή μπορεί να θεωρηθεί ικανοποιητικη· αν είναι πολύ μεγαλύτερα, τότε είναι πιθανό να υπάρχουν καλύτερα μονοπάτια.

2.1.2.2 Greedy Algorithm

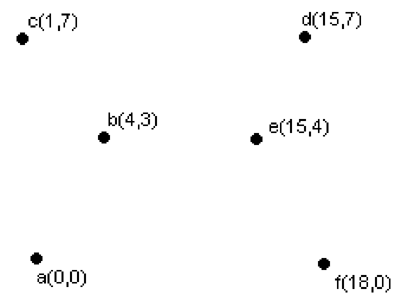
Πολλοί συγγραφείς ονομάζουν ως *Greedy* τον αλγόριθμο του πλησιέστερου γείτονα, αλλά είναι καταλληλότερη ονομασία για τον αλγόριθμο που περιγράφεται ως εξής:

> Αρχικά θεωρούμε ένα στιγμιότυπο ενός ολοκληρωμένου γράφου με τις πόλεις να παριστάνονται σα κορυφές με απόσταση $d(c_i , c_j)$ ανάμεσα σε κάθε ζεύγος πόλεων $\{c_i , c_j\}$. Στην περίπτωση αυτή το μονοπάτι είναι ο Hamiltonian κύκλος[1] του γράφου, π.χ., μια συνδεδεμένη συλλογή από κορυφές με κάθε πόλη να έχει βαθμό 2. Φτίαχνουμε τον κύκλο προχωρώντας μια κορυφή τη φορά, ξεκινώντας από τη κοντινότερη κορυφή, και προσθέτοντας κάθε φορά μια από τις πιο κοντινές κορυφές που έχουν απομείνει, μια κορυφή θεωρείται διαθέσιμη όταν δεν είναι ήδη μέρος της διαδρομής και αν την προσθέσουμε δε θα αυξήσουμε το βαθμό της κορυφής, που την προσθέτουμε, σε 3 ή αν δε δημιουργεί κύκλο μικρότερου μεγέθους από τον αριθμό των κορυφών του γράφου N.

[1] Hamiltonian κύκλος είναι ένας κύκλος σε έναν μη κατευθυνόμενο γράφο που επισκέπτεται κάθε κορυφή ακριβώς μια φορά και επιστρέφει στην αρχική κορυφή.

Όλο αυτό μπορεί να φανεί καλύτερα με ένα παράδειγμα:

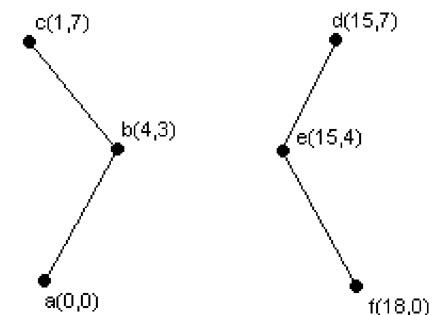
Στο *Σχήμα 2.3* βλέπουμε 6 πόλεις μαζί με τις συντεταγμένες τους.Οι αποστάσεις μεταξύ των πόλεων είναι {((*d,e*),*3*) , ((*b,c*),*5*) , ((*a,b*),*5*) , ((*e,f*),*5*) , ((*a,c*),*7.08*) , ((*d,f*), $\sqrt{58}$) , ((*b,e*), $\sqrt{22}$) , ((*b,d*), $\sqrt{137}$) , ((*c,d*),*14*) ,… ((*a,f*),*18*)}.



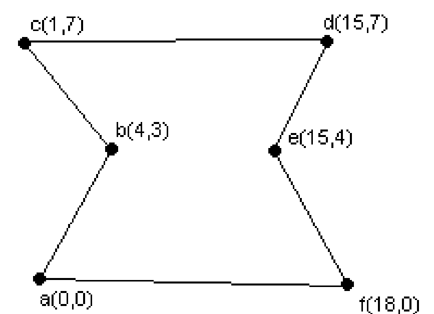
Σχήμα 2.3

Ο αλγόριθμος δουλεύει ως εξής:

επιλέγουμε (*d,e*)
επιλέγουμε (*a,b*)
επιλέγουμε (*b,c*)
επιλέγουμε (*e,f*)
δεν επιλέγουμε (*a,c*), δημιουργεί κύκλο με τα (*a,b*) και (*b,c*)
δεν επιλέγουμε (*d,f*), δημιουργεί κύκλο με τα (*d,e*) και (*e,f*)
δεν επιλέγουμε (*b,e*), γιατί κάνει το βαθμό του *b* ίσο με 3
δεν επιλέγουμε (*b,d*), γιατί κάνει το βαθμό του *b* ίσο με 3
επιλέγουμε (*c,d*)
.
.
.
επιλέγουμε (*a,f*)



Σχήμα 2.4



Σχήμα 2.5

Και έχουμε την τελική διαδρομή που φαίνεται στο *Σχήμα 2.5*.

Ο αλγόριθμος *Greedy* είναι πιο αργός από τον NN αλλά στην περίπτωση που μας δώσει τη χειρότερη διαδρομή, που μπορούμε να ακολουθήσουμε, η διαδρομή αυτή είναι καλύτερη από την αντίστοιχη του NN.

2.1.2.3 Nearest Insertion Algorithm

Ο τελευταίος αλγόριθμος που θα εξετάσουμε είναι ο *Nearest Insertion (NI)* ο οποίος επιλέγει επαναληπτικά τμήματα της διαδρομής k κόμβων και καθορίζει ποιός από τους εναπομείναντες n-k κόμβους θα είναι ο επόμενος που θα συμπεριληφθεί στη διαδρομή (βήμα επιλογής) και που θα τοποθετηθεί (βήμα εισαγωγής). Τα βήματα του αλγορίθμου αυτού είναι:

1. Ξεκινάμε με ένα τμήμα του γράφου που περιέχει μόνο τον κόμβο i.
2. Βρίσκουμε ένα κόμβο r τέτοιο ώστε c_{ir} (όπου c η απόσταση των κόμβων) να είναι το ελάχιστο και δημιουργούμε το μονοπάτι i-r-i που αποτελεί τμήμα της τελικής διαδρομής.
3. (Βήμα επιλογής) Δοσμένου του μονοπατιού, που περιγράφτηκε στο βήμα 2, βρίσκουμε έναν κόμβο r, που δεν αποτελεί μέρος του μονοπατιού, ο οποίος είναι ο πιο κοντινός σε έναν κόμβο j του μονοπατιού· π.χ. με τη μικρότερη c_{ri}.
4. (Βήμα εισαγωγής) Βρίσκουμε το τμήμα (i,j) στο μονοπάτι το οποίο ελαχιστοποιεί την σχέση $c_{ir} + c_{ri} - c_{ij}$. Εισάγουμε τον κόμβο r ανάμεσα στους i και j.
5. Αν όλοι οι κόμβοι αποτελούν μέρος της διαδρομής σταματάμε, διαφορετικά επιστρέφουμε στο βήμα 3.

Η περίπτωση της χειρότερης διαδρομής δίνεται από τον τύπο:

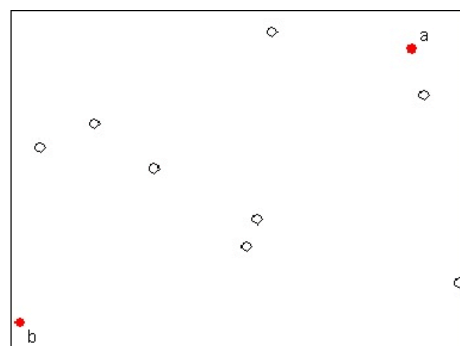
$$\frac{length_of_nearest_insertion_tour}{length_of_optimal_tour} \leq 2$$

2.1.3 Αλγόριθμοι που χρησιμοποιήθηκαν

Αφού παρουσιάσαμε αλγορίθμους που μπορούν να χρησιμοποιηθούν γενικά για την επίλυση του TSP ήρθε η ώρα να παρουσιάσουμε τους αλγορίθμους που χρησιμοποιήσαμε στην εργασία αυτή.

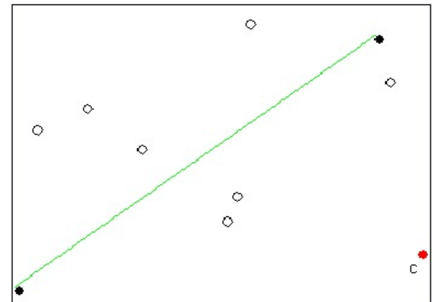
Αρχικά για την κατασκεύη της διαδρομής επιλέξαμε τον αλγόριθμο *Furthest Insertion* [9] ο οποίος είναι παρόμοιος με τον *Nearest Insertion* με τη διαφορά ότι επιλέγουμε τον κόμβο που έχει τη μέγιστη απόσταση αντί για την ελάχιστη. Τα βήματα του αλγορίθμου φαίνονται παρακάτω:

1. Βρίσκουμε τις 2 πόλεις με τη μεγαλύτερη μεταξύ τους απόσταση (*Σχήμα 2.6*), ας τις ονομάσουμε πόλη ‘a’ και πόλη ‘b’, και τις ενώνουμε μεταξύ τους (*Σχήμα 2.7*).
2. Επιλέγουμε την πόλη ‘c’ η οποία έχει τη μεγαλύτερη απόσταση από την γραμμή που ενώνει τις πόλεις ‘a’ και ‘b’(*Σχήμα 2.7*).



Σχήμα 2.6

Η πόλη 'c' εισάγεται στη διαδρομή ενώνοντάς την με κάθε μία από τις πόλεις 'a' και 'b', δημιουργώντας έτσι ένα τμήμα της τελικής διαδρομής (*Σχήμα 2.8*).



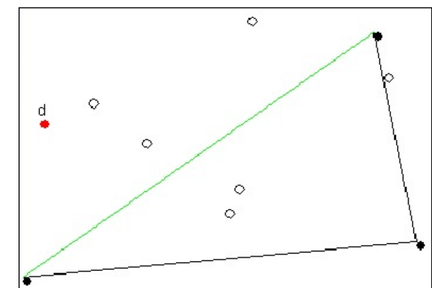
Σχήμα 2.7

3. Οι υπόλοιπες πόλεις εισάγονται ως εξής:

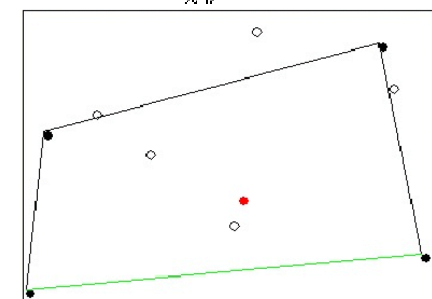
    i. Βρίσκουμε την πόλη (πόλη 'd') με τη μεγαλύτερη απόσταση από κάθε σημείο του μονοπατιού που έχουμε δημιουργήσει μέχρι τώρα (*Σχήμα 2.8*).

    ii. Σημειώνουμε την κορυφή από την οποία η πόλη 'd' έχει τη μικρότερη απόσταση (*Σχήμα 2.8*).

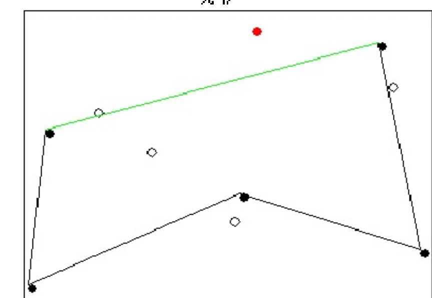
    iii. Διαγράφουμε την πλευρά αυτή και δημιουργούμε δύο νέες πλευρές ανάμεσα στην πόλη 'd' και σε κάθε μια από τις πόλεις που ένωνε η πλευρά που διαγράψαμε (*Σχήμα 2.9*).

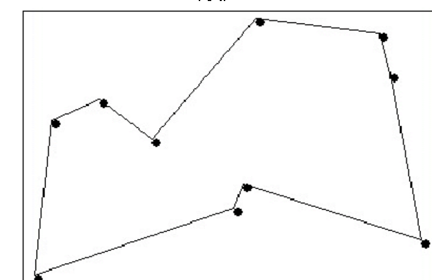


Σχήμα 2.8


Σχήμα 2.9

4. Επαναλαμβάνουμε το βήμα 3 μέχρι να συμπεριληφθούν όλες οι πόλεις στη διαδρομή (*Σχήματα 2.10, 2.11*).

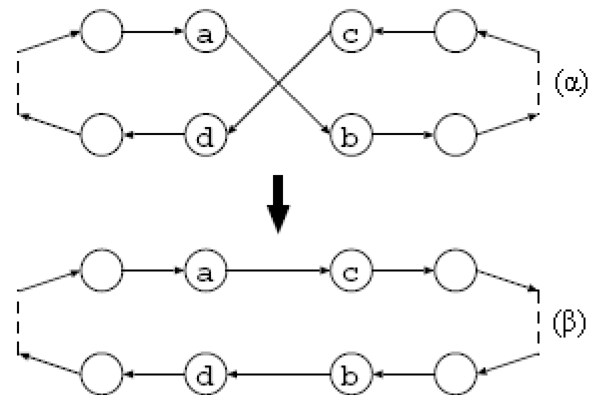

Σχήμα 2.10


Σχήμα 2.11

Στον *αλγόριθμο Furthest Insertion* η σχέση που μας δίνει την περίπτωση της χειρότερης διαδρομής είναι:

$$\frac{length_of_farthest_insertion_tour}{length_of_optimal_tour} \leq 2\ln(n) + 0.16$$

Για να βελτιστοποιήσουμε τη διαδρομή που δημιουργήσαμε, με χρήση της παραπάνω μέθοδου, χρησιμοποιήσαμε τον αλγόριθμο *2-opt*. Ο τρόπος που δουλεύει ο αλγόριθμος αυτός θα εξηγηθεί με ένα παράδειγμα. Έστω ότι έχουμε τη διαδρομή που φαίνεται στο *Σχήμα 2.12(α)*, αρχικά αφαιρούμε τις πλευρές $\overline{ab}$ και $\overline{cd}$ και στη συνέχεια αντιστρέφουμε τη σειρά που επισκεπτόμαστε τις κορυφές του ενός τμήματος της διαδρομής και εισάγουμε τις πλευρές $\overline{ac}$ και $\overline{bd}$ (*Σχήμα 2.12(β)*). Η διαδρομή που προκύπτει είναι μικρότερη από την αρχική γιατί $\overline{ab}+\overline{cd} > \overline{ac}+\overline{bd}$. Επαναλαμβάνουμε τη διαδικασία αυτή μέχρι να μην έχουμε περαιτέρω βελτιστοποίηση της αρχικής διαδρομής.



Σχήμα 2.12

Στο σημείο αυτό πρέπει να πούμε ότι ο αλγόριθμος *Farthest Insertion* επιλέχθηκε λόγω της ευκολίας στην υλοποίηση του και ο αλγόριθμος *2-opt* είναι ο αλγόριθμος που μας ζητήθηκε να βελτιστοποιήθει σε υλικό με χρήση των εργαλείων της Stretch. Περισσότερες λεπτομέρειες για την προσπάθεια βελτιστοποίησης, τα αποτελέσματά της και τα εργαλεία της Stretch θα δούμε στα κεφάλαια που ακολουθούν.

2.1.4 Εφαρμογές του TSP

Ολοκληρώνοντας την περιγραφή του προβλήματος του πλανόδιου πωλητή είναι σημαντικό να αναφέρουμε ορισμένες από τις εφαρμογές του. Εκτός από το σχεδιασμό των βέλτιστων διαδρομών για πλανόδιους πωλητές, ο αλγόριθμος αυτός έχει χρησιμοποιηθεί για εργασίες όπως ο σχεδιασμός των μετακινήσεων των συσκευών αυτόματης διάτρησης καρτών ηλεκτρονικών κυκλωμάτων και των μηχανισμών εφοδιασμού στους ορόφους των καταστημάτων. Επίσης ο TSP χρησιμοποιείται στον υπολογισμό των ακολουθιών DNA και στην εύρεση της βέλτιστης διάταξης (layout) κυκλωμάτων VLSI. Τέλος αξίζει να σημειωθεί μια ακόμα εφαρμογή του TSP είναι η βελτίωση των λογισμικών που χρησιμοποιούνται ώστε να είναι επιτυχής η παιδαγωγική μέθοδος E-learning.

2.2 Το πρόβλημα ελέγχου της ικανοποιησιμότητας μιας λογικής πρότασης (Boolean Satisfiability Problem, SAT)

2.2.1 Εισαγωγή

Το πρόβλημα της ικανοποιησιμότητας (satisfiability) είναι ένα πρόβλημα το οποίο καθορίζει αν μπορούν να γίνουν αναθέσεις τιμών στις μεταβλητές μιας Boolean συνάρτησης f με τρόπο τέτοιο ώστε η f να είναι ΑΛΗΘΗΣ. Εξίσου σημαντικό είναι να μπορούμε να πούμε ότι δεν υπάρχουν τιμές οι οποίες να ικανοποιούν την συνάρτηση, αυτό συνεπάγεται ότι η f θα είναι ΨΕΥΔΗΣ για όλες τις πιθανές τιμές των μεταβλητών της. Στην τελευταία περίπτωση λέμε ότι η συνάρτηση δεν ικανοποιείται (unsatisfiable), διαφορετικά ικανοποιείται (satisfiable). Το SAT είναι ένα NP-complete πρόβλημα (όπως και το πρόβλημα του πλανόδιου πωλητή που εξετάσαμε στην ενότητα 2.1). Επιπρόσθετα, το SAT είναι ένα πρόβλημα με μεγάλη σημασία για διάφορους τομείς της επιστήμης των υπολογιστών, όπως για παράδειγμα την τεχνητή νοημοσύνη, τη σχεδίαση hardware, τους αλγορίθμους και την επαλήθευση. Ως αποτέλεσμα των προηγούμενων, το θέμα πρακτικών SAT solvers (εφαρμογές που επιλύουν το SAT) έχει προσελκύσει μεγάλο ενδιαφέρον και πολλοί αλγόριθμοι έχουν προταθεί και εφαρμοστεί όπως π.χ. οι GRASP, POSIT, SATO, CHAFF, WalkSAT και MINISAT.

2.2.2 Προδιαγραφές Προβλήματος

Οι περισσότεροι solvers εφαρμόζονται σε προβλήματα στα οποία η Boolean συνάρτηση f βρίσκεται σε CNF μορφή. Η μορφή αυτή αποτελείται από το λογικό AND ενός ή περισσοτέρων *clauses* (προτάσεις), οι οποίες αποτελούνται από το λογικό OR ενός ή περισσοτέρων *literals* (μεταβλητές). Το *literal* αποτελεί την θεμελιώδη λογική μονάδα στο πρόβλημα και ουσιαστικά είναι το στιγμιότυπο μιας μεταβλητής ή του συμπληρώματός της. Όλες οι Boolean συναρτήσεις μπορούν να γραφτούν σε CNF μορφή. Το προτέρημα της CNF μορφής είναι ότι για να είναι η f αληθής θα πρέπει κάθε μια από τις επιμέρους *clauses* να είναι αληθής.

2.2.3 MINISAT

2.2.3.1 Εισαγωγή

Όπως αναφέραμε στην ενότητα 2.2.1 υπάρχουν πολλοί Sat solvers. Ένας από τους πιο σύγχρονους είναι ο MINISAT ο οποίος και σημείωσε επιτυχία στον διαγωνισμό για SAT solvers το 2005. Η δημιουργία του MINISAT ξεκίνησε το 2003 στο MIT από τους Niklas Eén και Niklas Sörensson σαν μια προσπάθεια να βοηθήσουν τους ανθρώπους να ενταχθούν στην κοινότητα του SAT με έναν μικρό, αλλά ταυτόχρονα αποτελεσματικό, SAT solver ο οποίος συνοδεύεται από καλή βιβλιογραφία για την κατανόησή του.

Η πρώτη του έκδοση ήταν λίγο περισσότερο από 600 γραμμές κώδικα, χωρίς να συμπεριλάβουμε τα σχόλια και τις κενές γραμμές, ενώ ταυτόχρονα περιείχε όλα τα χαρακτηριστικά των, μέχρι το 2003, δημοφιλών SAT solvers. Με την πάροδο του χρόνου ο κώδικας αυτός αυξήθηκε για να συμπεριλάβει ορισμένες βελτιστοποιήσεις, αλλά παραμένει μικρός και κατανοητός σε σχέση με τους κώδικες άλλων SAT solvers.

Ορισμένα από τα κύρια χαρακτηριστικά του MINISAT είναι:

- Η δυνατότητα του να τροποποιηθεί.
- Η υψηλή αποδοτικότητά του.
- Η σχεδίασή του με τρόπο που του επιτρέπει την εύκολη ενσωμάτωσή του σε άλλους κώδικες-προβλήματα.

2.2.3.2 Περίληψη του SAT-solver

Για να κατανοήσουμε τον MINISAT θα πρέπει πρώτα να μπορέσουμε να καταλάβουμε τη γενική περίπτωση του *conflict-driven SAT-solver* (SAT-solver οδηγούμενου-από-συγκρούσεις) ώστε να περάσουμε ομαλά στα χαρακτηριστικά του MINISAT.

Ένας τυπικός conflict-driven SAT-solver λειτουργεί με clauses (οι οποίες αποτελούνται από δύο ή παραπάνω literals) και *assignments* (εντολές-αναθέσεις). Παρόλο που οι assignments μπορούν να θεωρηθούν unit-clauses (μοναδιαίες-προτάσεις), αντιμετωπίζονται με ξεχωριστό τρόπο και είναι καλύτερο να τις βλέπουμε σαν έναν ξεχωριστό τύπο πληροφορίας.

Σε έναν τυπικό solver ο μοναδικός μηχανισμός εξαγωγής αποτελεσμάτων βασίζεται στο *unit propagation* (μοναδιαία διάδοση). Αυτό σημαίνει ότι αμέσως μόλις μια clause γίνει *unit* (μοναδιαία) κάτω από τις τρέχουσες αναθέσεις τιμών (όλα τα literals εκτός από ένα είναι ψευδή), το μοναδικό literal που δεν έχει τιμή γίνεται αυτόματα αληθές, με τον τρόπο αυτό είναι πιθανόν να κάνουμε unit περισσότερες από μια clauses. Η διαδικασία αυτή επαναλαμβάνεται μέχρι να μη μπορεί να διαδοθεί περαιτέρω πληροφορία.

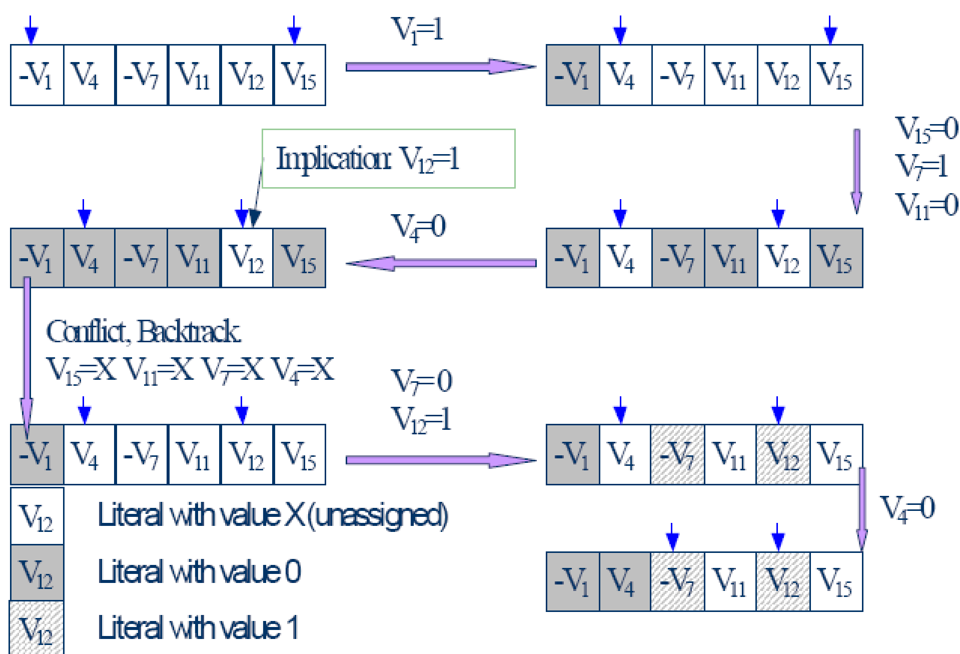
Η πιο σύνθετη διαδικασία ενός μοντέρνου SAT solver είναι η διαδικασία αναζήτησης. Πιο συγκεκριμένα επιλέγονται μεταβλητές και τους αναθέτονται τιμές (γίνονται δηλαδή υποθέσεις), μέχρις ότου η διαδικασία της διάδοσης να συναντήσει *conflict* (σύγκρουση, δηλάδη όταν όλα τα literals μιας clause γίνουν ψευδή). Στο σημείο αυτό παράγεται και εισάγεται στο SAT πρόβλημα μια *conflict clause* (κανόνας σύγκρουσης) και ότι υποθέσεις είχαμε κάνει μέχρι τώρα αναιρούνται ώσπου η πρόταση που προκάλεσε τη σύγκρουση να γίνει unit, όταν αυτό επιτευχθεί η διαδικασία αναζήτησης συνεχίζεται κανονικά.

Στην περίπτωση του MINISAT μπορούμε να επεκτείνουμε τους περιορισμούς με την προσθήκη αυθαίρετων περιορισμών. Κάτι τέτοιο έχει ως αποτέλεσμα ότι ο solver θα πρέπει να είναι σε θέση να αποθηκεύει αυτούς τους περιορισμούς, να εξάγει μοναδιαία πληροφορία εφαρμόζοντάς τους και να δημιουργεί conflict clauses από αυτούς.

Παρακάτω θα δούμε τα βασικά χαρακτηριστικά λειτουργίας του MINISAT.

2.2.3.2.1 Διάδοση (propagation)

Κατά την διαδικασία της διάδοσης (απόδοσης τιμών στα literals) στο MINISAT για κάθε literal κρατάμε μια λίστα περιορισμών, οι οποίοι μπορεί να παράγουν μοναδιαία πληροφορία αν το literal γίνει αληθές. Για την περίπτωση των clauses δε μπορεί να παραχθεί μοναδιαία πληροφορία μέχρις ότου όλα τα literals εκτός από ένα να γίνουν ψευδή. Επίσης επιλέγουμε δύο τυχαία literals p και q της clause, που δεν τους έχουμε αποδώσει ακόμα τιμή, και αναφορές στην clause προστίθενται στις λίστες των $\overline{p}$ και $\overline{q}$ αντίστοιχα. Τα literals αυτά λέμε ότι είναι *watched* (παρακολουθούμενα) και οι λίστες με τους περιορισμούς αναφέρονται ως *watcher lists* (λίστες παρακολούθησης). Όταν ένα watched literal γίνει αληθές, μπαίνει σε λειτουργία ο περιορισμός ώστε να δούμε αν μπορούμε να διαδόσουμε την πληροφορία ή αν θα πρέπει να παρακολουθήσουμε δύο νέα literals χωρίς τιμές. Όλη αυτή η διαδικασία των watched literals φαίνεται στο *Σχήμα 2.13*



Σχήμα 2.13

Ένα χαρακτηριστικό του συστήματος παρακολούθησης για τις clauses είναι ότι όταν θα χρειαστεί να κάνουμε ορισμένα βήματα προς τα πίσω και άρα κάποιες αναιρέσεις (*backtracking*) δε χρειάζεται να γίνουν επιμέρους προσαρμογές στις watcher lists. Αυτό κάνει το backtracking πολύ "φθηνό" στην υλοποίησή του. Όμως, για διαφορετικούς τύπους περιορισμών η μέθοδος αυτή δεν είναι απαραίτητα καλή προσέγγιση. Έτσι το MINISAT υποστηρίζει την προαιρετική χρήση *undo lists*, δηλαδή λίστες οι οποίες αποθηκεύουν ποιοί περιορισμοί πρέπει να ενημερωθούν όταν μια μεταβλητή δεν έχει τιμή μετά από το backtracking.

2.2.3.2.2 Learning (Εκμάθηση)

Το MINISAT έχει την ικανότητα της εκμάθησης – εκπαίδευσης. Η διαδικασία εκμάθησης ξεκινά όταν ένας περιορισμός προκαλέσει σύγκρουση (δηλαδή όταν ο περιορισμός είναι αδύνατον να ικανοποιηθεί) με την τρέχουσα τιμή που του έχει ανατεθεί. Στην περίπτωση αυτή, ο περιορισμός που προκάλεσε τη σύγκρουση καλείται να δώσει ένα σετ από τιμές για τις μεταβλητές οι οποίες αναιρούν την εξαίρεση, αν αναφερόμαστε σε clauses τότε εννοούμε όλα τα literals της clause (που είναι ψευδή κατά την σύγκρουση). Κάθε ανάθεση τιμών που επιστρέφεται θα πρέπει να είναι είτε υπόθεση της διαδικασίας εκμάθησης είτε το αποτέλεσμα της διάδοσης (propagation) κάποιου περιορισμού. Στη δεύτερη περίπτωση οι περιορισμοί που παράχθηκαν πρέπει να δώσουν με τη σειρά τους το σετ τιμών που προκάλεσαν τη διάδοση να συμβεί συνεχίζοντας έτσι το backtracking. Η διαδικασία συνεχίζεται μέχρι να εκπληρωθεί κάποια συνθήκη. Με τον τρόπο αυτό καταλήγουμε

σε ένα σετ τιμών που προκαλούν σύγκρουση και μια clause, που απαγορεύει αυτή την ανάθεση (που θα δημιουργήσει εξαίρεση), προστίθεται στη clause database.

Οι clauses που έχουν προκύψει από τη διαδικασία της εκμάθησης (*learnt clauses*) εξυπηρετούν δύο σκοπούς: οδηγούν το backtracking και επιταχύνουν μελλοντικές συγκρούσεις "αποθηκεύοντας" την αιτία της σύγκρουσης. Κάθε clause αποτρέπει μόνο ένα σταθερό αριθμό από αποτελέσματα, αλλά καθώς οι καταγεγραμμένες clauses αυξάνονται και συμμετέχουν στη διάδοση των τιμών, το αποτέλεσμα της συσσώρευσής τους μπορεί να είναι πολύ σημαντικό. Όμως, καθώς ο αριθμός των learnt clauses αυξάνεται η διάδοση των τιμών γίνεται όλο και πιο αργή, για το λόγο αυτό ο αριθμός τους μειώνεται περιοδικά κρατώντας μόνο τις clauses που φαίνεται να είναι χρήσιμες με βάση κάποια ευριστική μέθοδο.

2.2.3.2.3 Search (Εύρεση)

Η διαδικασία της εύρεσης στο MINISAT και γενικότερα στους conflict-driven SAT solvers θα μπορούσε να περιγραφεί αναδρομικά ως μια πιο εκλεπτυσμένη λύση αλλά συνήθως περιγράφεται επαναληπτικά. Πιο συγκεκριμένα ξεκινάμε με την επιλογή μιας μεταβλητής που δεν έχει τιμή, ας πούμε x (την οποία και ονομάζουμε μεταβλητή απόφασης – *decision variable*) και της αναθέτουμε μια τιμή, για παράδειγμα θεωρούμε ότι είναι αληθής. Οι συνέπειες του x = αληθής παράγονται και μπορούν να οδηγήσουν σε αναθέσεις περισσότερων μεταβλητών. Όλες οι αναθέσεις στις

μεταβλητές που συμβαίνουν σαν επακόλουθο του *x* λέμε ότι ανήκουν στο ίδιο επίπεδο απόφασης (*decision level*), ξεκινώντας την αρίθμηση των επιπέδων από το 1 για την πρώτη υπόθεση που κάναμε και αυξάνοντάς την σταδιακά. Αναθέσεις που έγιναν πριν την πρώτη υπόθεση (επίπεδο απόφασης 0) ονομάζονται *top-level.*

Όλες οι αναθέσεις αποθηκεύονται σε μια στοίβα με τη σειρά με την οποία έγιναν η οποία ονομάζεται *trail*. Το trail χωρίζεται σε επίπεδα απόφασης και χρησιμοποιείται για να πραγματοποιήσουμε αναιρέσεις σε περίπτωση backtracking.

Η φάση της απόφασης συνεχίζεται ώσπου όλες οι μεταβλητές να έχουν τιμή, στην περίπτωση αυτή προκύπτει ένα μοντέλο ή μια σύγκρουση. Σε περίπτωση σύγκρουσης καλείται η διαδικασία εκμάθησης και παράγεται μια conflict clause. Το trail χρησιμοποιείται για να πραγματοποιήσει τις αναιρέσεις, προχωρώντας ένα επίπεδο κάθε φορά, μέχρι ακριβώς ένα από τα literals της learnt clause να μην έχει τιμή (όλα είναι ψευδή τη στιγμή που συμβαίνει η σύγκρουση). Από κατασκευής, η conflict clause δε μπορεί να μεταβεί απευθείας σε μια clause με δύο literals χωρίς τιμές. Αν η clause παραμείνει μοναδιαία για πολλά επίπεδα απόφασης τότε επιλέγουμε το χαμηλότερο επίπεδο απόφασης, η περίπτωση αυτή αναφέρεται ως *backjumping* ή *non-chronological backtracking*.

Μια σημαντική παράμετρος της διαδικασίας εύρεσης είναι ότι δίνουμε προτεραιότητα σε μεταβλητές που εμπλέχτηκαν σε πρόσφατες συγκρούσεις.

2.2.3.2.4 Activity (Δραστηριότητα)

Μια σημαντική τεχνική που χρησιμοποιούμε στο MINISAT είναι η δυναμική ταξινόμηση των μεταβλητών η οποία βασίζεται στο *activity* (δραστηριότητα) τους. Στο σημείο αυτό πρέπει να αναφέρουμε ότι στο MINISAT δεν κάνουμε κανένα διαχωρισμό μεταξύ p και $\overline{p}$.

Κάθε μεταβλητή έχει ένα activity προσαρμοσμένο σε αυτή. Κάθε φορά που η μεταβλητή εμφανίζεται σε μια καταγεγραμμένη clause που προκαλεί σύγκρουση το activity της αυξάνεται, η διαδικασία αυτή ονομάζεται *bumping*. Αφού καταγραφεί η σύγκρουση το activity όλων των μεταβλητών του συστήματος πολλαπλασιάζεται με μια σταθερά μικρότερη του 1, με τον τρόπο αυτό το activity των μεταβλητών φθήνει με το χρόνο. Το τρέχον άθροισμα καθορίζει το activity μιας μεταβλητής.

Επίσης στο MINISAT έχουμε μια παρόμοια ιδέα για τις clauses, κάθε φορά που μια learnt clause χρησιμοποιείται στην ανάλυση μιας σύγκρουσης το activity της μεταβάλλεται. Clauses που είναι ανενεργές (μικρό activity) περιοδικά απομακρύνονται.

2.2.3.2.5 Constraint removal (Απομάκρυνση περιορισμών)

Η βάση δεδομένων που περιέχει τους περιορισμούς διαιρείται σε δύο τμήματα: τους περιορισμούς του προβλήματος και τις learnt clauses. Όπως

έχουμε ήδη πει το σετ των learnt clauses μειώνεται περιοδικά για να αυξηθεί η απόδοση της διάδοσης. Οι learnt clauses χρησιμοποιούνται ώστε να "προλαμβάνουμε" νέες clauses που θα περιληφθούν στο δέντρο εύρεσης πράγμα το οποίο θα έχει σαν αποτέλεσμα ένα μεγάλο χώρο στον οποίο θα ψάχνουμε για τους περιορισμούς του προβλήματος. Στο MINISAT ξεκινάμε από ένα μικρό σετ από learnt clauses και αυξάνουμε το μέγεθος. Όταν αφαιρούμε learnt clauses θα πρέπει να προσέχουμε ώστε να μην αφαιρούμε clauses η οποιές χρησιμοποιούνται στο τρέχον backtracking. Οι περιορισμοί του προβλήματος απομακρύνονται αν ικανοποιούνται στο top-level.

2.2.3.2.6 Top-level solver (επίλυση στο επίπεδο απόφασης 0)

Για την περίπτωση που βρισκόμαστε στο επίπεδο 0 χρησιμοποιούμε επανεκκινήσεις ώστε να ξεφύγουμε από τα άσκοπα μέρη του δέντρου εύρεσης ενώ ταυτόχρονα μεταβάλουμε τον αριθμό των learnt clauses που κρατάμε σε μια δεδομένη χρονική στιγμή.

2.3 Γενετικός Αλγόριθμος (Genetic Algorithm, GA)

2.3.1 Εισαγωγή και Ιστορικά Στοιχεία

Ο γενετικός αλγόριθμος (GA) είναι μια τεχνική η οποία χρησιμοποιείται στην επιστήμη υπολογιστών για να βρούμε τις ακριβείς ή προσεγγιστικές λύσεις σε συστήματα που μπορούν να περιγραφούν ως μαθηματικά προβλήματα. Είναι χρήσιμος σε προβλήματα που περιέχουν πολλές παραμέτρους/διαστάσεις και δεν υπάρχει αναλυτική μέθοδος που να μπορεί να βρει το βέλτιστο συνδυασμό τιμών για τις μεταβλητές ώστε το υπό εξέταση σύστημα να αντιδρά με τον όσο το δυνατό θεμιτό τρόπο. Οι γενετικοί αλγόριθμοι αποτελούν μια ειδική κατηγορία εξελικτικών αλγορίθμων οι οποίοι χρησιμοποιούν τεχνικές εμπνευσμένες από την εξελικτική βιολογία, όπως για παράδειγμα η κληρονομικότητα, η γενετική μετάλλαξη, η φυσική επιλογή και η διασταύρωση.

Η προσομοίωση της εξέλιξης στους υπολογιστές ξεκίνησε το 1954 με τη δουλεία του Nils Aall Barricelli στο Πρίνστον, η δουλειά του οποίου όμως δεν τράβηξε την απαραίτητη προσοχή. Το 1957 ο Αυστραλός γενετιστής Alex Fraser εκδίδει μια σειρά από συγγράμματα σε σχέση με την τεχνητή επιλογή των οργανισμών. Μετά από αυτή την αρχή η προσομοίωση της εξέλιξης σε υπολογιστές από βιολόγους έγινε όλο και πιο συχνή και οι μέθοδοι που χρησιμοποιήθηκαν περιέχονται σε βιβλία των Fraser και Burnell (1970) και Crosby (1973). Οι προσομοιώσεις του Fraser περιείχαν όλα τα θεμελιώδη στοιχεία των μοντέρνων γενετικών

αλγορίθμων. Επιπρόσθετα, στη δεκαετία του 1960 εκδόθηκαν συγγράμματα του Hans Bremermann τα οποία περιείχαν λύσεις σε προβλήματα βελτιστοποίησης χρησιμοποιώντας τις αρχές τις εξέλιξης. Άλλοι αξιοσημείωτοι πρωτοπόροι των γενετικών αλγορίθμων είναι οι Richard Friedberg, George Friedman και Michael Conrad.

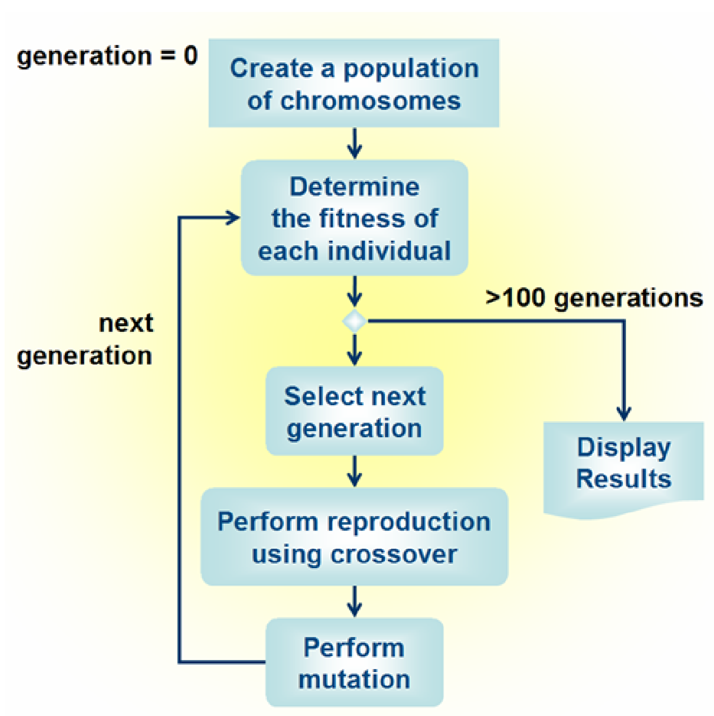
Οι γενετικοί αλγόριθμοι όμως, έγιναν ευρέως γνωστοί στις αρχές τις δεκαετίας του 1970 από τη δουλειά του John Holland και συγκεκριμένα από το βιβλίο του *Adaptation in Natural and Artificial Systems* (1975). Η έρευνα για τους γενετικούς αλγορίθμους παρέμεινε θεωρητική ως και τα μέσα της δεκαετίας του 1980 όπου πραγματοποιήθηκε το πρώτο παγκόσμιο συνέδριο για γενετικούς αλγορίθμους στο Πίτσμπουργκ της Πενσυλβανίας.

Καθώς το ακαδημαϊκό ενδιαφέρον μεγάλωνε, η δραματική αύξηση της υπολογιστικής δύναμης των υπολογιστών επέτρεψε την πρακτική εφαρμογή των γενετικών αλγορίθμων. Στα τέλη του 1980 η General Electric άρχισε να πουλάει το πρώτο παγκόσμιο προϊόν γενετικού αλγορίθμου. Το 1989 η εταιρεία Axcelis, Inc. παρουσίασε το δεύτερο GA προϊόν και το πρώτο για επιτραπέζιους υπολογιστές.

2.3.2 Μεθοδολογία

Οι γενετικοί αλγόριθμοι υλοποιούνται ως προσομοίωση σε υπολογιστές στην οποία ένας πληθυσμός από αφηρημένες αναπαραστάσεις (οι οποίες καλούνται *άτομα* ή *γενότυποι* και τα μέλη τους *χρωμοσώματα*) των

υποψήφιων λύσεων, ενός προβλήματος βελτιστοποίησης, εξελίσετε σε καλύτερες λύσεις. Παραδοσιακά οι λύσεις παριστάντονται με δυαδικό τρόπο σαν μια αλληλουχία 0 και 1, αλλά άλλες κωδικοποιήσεις είναι επίσης πιθανές. Η εξέλιξη συνήθως ξεκινάει με τη δημιουργία ενός πληθυσμού τυχαίων ατόμων και συμβαίνει σε γενιές. Σε κάθε γενιά η ικανότητα (*fitness*) κάθε μέλους του πληθυσμού αποτιμάται, ένας αριθμός ατόμων (βασισμένος στην ικανότητα) επιλέγεται από τον τρέχων πληθυσμό και τροποποιείται (με γενετικό ανασυνδυασμό και πιθανόν με τυχαία μετάλλαξη) ώστε να δημιουργήσει ένα νέο πληθυσμό. Ο νέος πληθυσμός χρησιμοποιείται στην επόμενη επανάληψη του αλγορίθμου. Συνήθως ο αλγόριθμος τερματίζεται όταν παράγεται ο μέγιστος αριθμός γενεών ή όταν φτάσουμε σε ένα ικανοποιητικό επίπεδο ικανότητας για τον πληθυσμό. Όταν ο αλγόριθμος τερματίζεται εξαιτίας του μέγιστου αριθμού γενεών είναι πιθανό να έχουμε ή να μην έχουμε φτάσει σε μια ικανοποιητική λύση. Όλα τα παραπάνω συνοψίζονται στο *Σχήμα 2.14*.



Σχήμα 2.14

Ένας τυπικός γενετικός αλγόριθμος απαιτεί να καθοριστούν δύο πράγματα:

1. μια γενετική αναπαράσταση των πιθανών λύσεων του προβλήματος,
2. μια συνάρτηση υπολογισμού της ικανότητας για αξιολόγηση των μελών του πληθυσμού.

Μια τυπική αναπαράσταση της λύσης είναι μια διάταξη από bits, αλλά και διατάξεις διαφορετικών τύπων μπορούν να χρησιμοποιηθούν ουσιαστικά με τον ίδιο τρόπο. Η βασική ιδιότητα των γενετικών αλγορίθμων είναι ότι η διαδικασία της διασταύρωσης γίνεται εύκολα εξαιτίας του σταθερού μεγέθους των διατάξεων. Είναι δυνατό να έχουμε και διατάξεις μεταβλητού μεγέθους αλλά αυτό οδηγεί σε αύξηση της δυσκολίας της διασταύρωσης. Η συνάρτηση υπολογισμού της ικανότητας ορίζεται για κάθε άτομο, μετράει την *ποιότητα* της κάθε λύσης και εξαρτάται κάθε φορά από τα δεδομένα του προβλήματος.

Από τη στιγμή που έχουμε ορίσει το είδος της γενετικής αναπαράστασης και τη συνάρτηση υπολογισμού της ικανότητας, ο γενετικός αλγόριθμος αρχικοποιεί έναν πληθυσμό τυχαίων λύσεων και αρχίζει να τον βελτιστοποιεί επαναληπτικά με διαδικασίες όπως η μετάλλαξη, η διασταύρωση και η επιλογή.

2.3.2.1 Επιλογή (Selection)

Από κάθε διαδοχική γενιά, επιλέγεται ένας αριθμός μελών του πληθυσμού ώστε να "γεννήσει" την επόμενη γενιά (*γονείς*). Τα μέλη άτομα που θα γίνουν γονείς επιλέγονται με διάφορες μεθόδους αποτίμησης της ικανότητας. Υπάρχουν μέθοδοι που υπολογίζουν την ικανότητα κάθε μέλους και στη συνέχεια επιλέγουν τις καλύτερες λύσεις, όπως υπάρχουν και μέθοδοι οι οποίες υπολογίζουν την ικανότητα ενός τυχαίου αριθμού μελών του πληθυσμού αφού η πρώτη μέθοδος καταναλώνει αρκετό χρόνο.

Οι περισσότερες συναρτήσεις επιλογής είναι στοχαστικές και σχεδιάζονται με τρόπο ώστε να μπορούμε να κρατήσουμε την ποικιλομορφία του πληθυσμού, αποτρέποντας τη γρήγορη σύγκλιση σε έναν "φτωχό" πληθυσμό. Δύο κλασικές μέθοδοι επιλογής είναι οι *ρουλέτα με σχισμές* (*roulette wheel selection* – τα μέλη του πληθυσμού τοποθετούνται στη ρουλέτα ανάλογα με την ικανότητά τους και μπορούν να επιλεχθούν περισσότερες από μια φορές κατά τη διάρκεια της επιλογής) και *tournament selection* (διαδικασία τυχαίας επιλογής από ένα σύνολο γονέων).

2.3.2.2 Αναπαραγωγή (Mutation, Crossover)

Το επόμενο βήμα είναι η δημιουργία μιας νέας γενιάς χρησιμοποιώντας τη διασταύρωση και/ή μετάλλαξη. Για κάθε νέα λύση που θα παραχθεί, έχει επιλεχθεί ένα ζεύγος γονέων. Με τις μεθόδους της διασταύρωσης και μετάλλαξης κάθε γονιός συνεισφέρει κάποια από τα χαρακτηριστικά του

στο παιδί. Για κάθε παιδί, νέοι γονείς επιλέγονται κάθε φορά και η διαδικασία συνεχίζεται μέχρι να έχουμε έναν νέο πληθυσμό κατάλληλου μεγέθους. Με τον τρόπο αυτό η νέα γενιά που προκύπτει είναι διαφορετική από την προηγούμενη και η μέση ικανότητα των ατόμων έχει αυξηθεί, εφόσον κάθε φορά μόνο οι καλύτεροι γονείς επιλέγονται για αναπαραγωγή.

2.3.2.3 Τερματισμός

Η διαδικασία αναπαραγωγής συνεχίζεται μέχρι να φτάσουμε σε κάποια συνθήκη τερματισμού. Συνηθισμένες συνθήκες τερματισμού είναι:

- Έχει επιτευχθεί μια λύση που ικανοποιεί τα ελάχιστα κριτήρια που θέσαμε στην αρχή του προβλήματος.
- Ένας συγκεκριμένος και προκαθορισμένος αριθμός γενεών έχει δημιουργηθεί.
- Η καλύτερη λύση με βάση την ικανότητα έχει δημιουργηθεί και επιπλέον επαναλήψεις δε μπορούν να παράγουν καλύτερα αποτελέσματα.
- Έχουμε καταναλώσει όλους τους πόρους μας (υπολογιστικό χρόνο/χρήμα).
- Διακοπή από το χρήστη.
- Συνδυασμός των παραπάνω.

2.3.2.4 Ψευδοκώδικας

Όλα τα παραπάνω μπορούν να περιγραφούν από τον ψευδοκώδικα που ακολουθεί:

1. Επιλέγουμε τον αρχικό πληθυσμό
2. Υπολογίζουμε την ικανότητα κάθε ατόμου του πληθυσμού
3. Επαναλαμβάνουμε
    1. Επιλέγουμε τα καλύτερα άτομα με βάση την ικανότητα
    2. Δημιουργούμε τη νέα γενιά με χρήση της διασταύρωσης και της μετάλλαξης και "γεννάμε" το παιδί
    3. Υπολογίζουμε την ικανότητα του παιδιού
    4. Αντικαθιστούμε το άτομα του πληθυσμού με τη χειρότερη ικανότητα με το παιδί
4. Ως τον τερματισμό

2.3.3 Πλεονεκτήματα και μειονεκτήματα γενετικών αλγορίθμων και εφαρμογές τους

Μερικά από τα σημαντικότερα πλεονεκτήματα που έχει η χρήση γενετικών αλγορίθμων για την επίλυση προβλημάτων είναι τα εξής:

1) Μπορούν να επιλύουν δύσκολα προβλήματα γρήγορα και αξιόπιστα.

2) Μπορούν εύκολα να συνεργαστούν με τα υπάρχοντα μοντέλα και συστήματα.

3) Είναι εύκολα επεκτάσιμοι και εξελίξιμοι.

4) Εφαρμόζονται σε πολύ περισσότερα πεδία από κάθε άλλη μέθοδο.

5) Δεν απαιτούν περιορισμούς στις συναρτήσεις που επεξεργάζονται.

6) Δεν ενδιαφέρει η σημασία της υπό εξέταση πληροφορίας.

7) Έχουν από τη φύση τους το στοιχείο του παραλληλισμού.

8) Είναι μία μέθοδος που κάνει ταυτόχρονα εξερεύνηση του χώρου αναζήτησης και εκμετάλλευση της ήδη επεξεργασμένης πληροφορίας.

Παρόλα τα θετικά στοιχεία των γενετικών αλγορίθμων υπάρχουν κάποιοι λόγοι που ίσως θα μπορούσαν να σταθούν εμπόδιο στην εξάπλωση αυτής της τεχνολογίας, όπως:

1) Προβλήματα εξοικείωσης με τη Γενετική.

2) Το πρόβλημα του χρόνου.

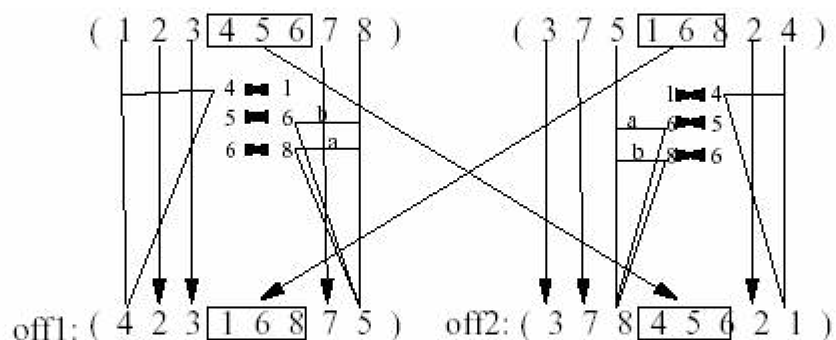
Όπως φαίνεται από όλα τα παραπάνω οι γενετικοί αλγόριθμοι είναι πολύ εύχρηστα εργαλεία και βρίσκουν εφαρμογή στην επιστημή των υπολογιστών, στα οικονομικά, στη χημεία, στα μαθηματικά, στη φυσική, στη βιοπληροφορική, στη φυλογενετική και σε άλλα πεδία.

2.3.4 Επίλυση του TSP με γενετικό αλγόριθμο

Όπως είδαμε στην προηγούμενη ενότητα οι γενετικοί αλγόριθμοι χρησιμοποιούνται σε πολλούς επιστημονικούς τομείς και επιλύουν διάφορα προβλήματα. Στην παρούσα εργασία επιλέξαμε να λύσουμε με γενετικό τρόπο το πρόβλημα του πλανόδιου πωλητή, που παρουσιάσαμε προηγουμένως, λόγω του ότι ήδη είχαμε το θεωρητικό υπόβαθρο που χρειάζεται για τη λύση του προβλήματος αυτού. Στη συνέχεια της ενότητας θα παρουσιάσουμε την υλοποίησή μας χωρίς όμως να μπούμε σε λεπτομέρειες του κώδικα.

Το πρώτο στάδιο του γενετικού αλγορίθμου όπως είδαμε είναι η δημιουργία του πληθυσμού. Στην περίπτωση του TSP, ο αρχικός πληθυσμός αποτελείται από διαδρομές, που θα μπορούσε να ακολουθήσει ο πωλητής, οι οποίες έχουν δημιουργηθεί με τυχαίο τρόπο. Χρησιμοποιώντας, όπως προηγουμένως, για είσοδο τη βιβλιοθήκη TSPLIB μπορούμε να υπολογίσουμε το βάρος κάθε διαδρομής, που στην προκειμένη περίπτωση είναι και η ικανότητα του κάθε ατόμου του πληθυσμού.

Στη συνέχεια επιλέγουμε τα άτομα με τη μικρότερη ικανότητα (όπως την υπολογίσαμε) ώστε να αποτελέσουν τους γονείς της επόμενης γενιάς. Η διαδικασία της διασταύρωσης θα εξηγηθεί με ένα παράδειγμα (*Σχήμα 2.15*). Έστω ότι έχουμε, ως υποψήφιους γονείς, τις διαδρομές (1 2 3 4 5 6 7 8) και (3 7 5 1 6 8 2 4). Αρχικά, στις δύο διαδρομές αυτές επιλέγουμε, με τυχαίο τρόπο, δύο σημεία τομής. Το τμήμα των διαδρομών που βρίσκεται ανάμεσα στα σημεία τομής αποτελεί τους κανόνες αντικατάστασης που θα χρησιμοποιηθούν αργότερα για τη δημιουργία των παιδιών.



Σχήμα 2.15

Στο παράδειγμά μας έχουμε τους εξής κανόνες αντικατάστασης:

4↔1, 5↔6, 6↔8

Στη συνέχεια το τμήμα ανάμεσα στα σημεία τομής του πρώτου γονιού αντιγράφεται στο δεύτερο παιδί και το αντίστοιχο τμήμα του δεύτερου γονιού αντιγράφεται στο πρώτο παιδί, έτσι έχουμε:

παιδί1: (x x x 1 6 8 x x) και παιδί2: (x x x 4 5 6 x x)

Στο επόμενο βήμα, συμπληρώνουμε τα στοιχεία που απομένουν στο παιδί 'ν' (όπου ν=1,2) αντιγράφοντας τα στοιχεία από τον ν-στο πατέρα. Σε περίπτωση που μια πόλη υπάρχει ήδη στο παιδί αντικαθίσταται σύμφωνα με τους κανόνες αντικατάστασης που έχουμε ορίσει. Έτσι, το πρώτο στοιχείο στο παιδί1 θα είναι 1 όπως και στον πρώτο πατέρα. Επειδή όμως το στοιχείο 1 έχει ήδη χρησιμοποιηθεί εξαιτίας του κανόνα αντικατάστασης 1↔4 το πρώτο στοιχείο του παιδιού1 είναι το 4, το δεύτερο, τρίτο και τέταρτο στοιχείο του παιδιού1 μπορούμε να τα πάρουμε από τον γονιό1. Ωστόσο, το τελευταίο στοιχείο του παιδιού1 θα πρέπει να είναι 8 το οποίο όμως χρησιμοποιείται, έτσι εξαιτίας των κανόνων αντικατάστασης 6↔6 και 6↔5 επιλέγεται να είναι το 5. Τελικά έχουμε:

παιδί1: (4 2 3 1 6 8 7 5) και ομοίως παιδί2: (3 7 8 4 6 5 2 1)

Εφόσον έχουμε ολοκληρώσει τη διασταύρωση περνάμε στη διαδικασία της μετάλλαξης την οποία για την εργασία αυτή υλοποιήσαμε με τη μέθοδο 2opt που περιγράψαμε στην ενότητα 2.1.3. Στο σημείο αυτό πρέπει να αναφέρουμε ότι η μετάλλαξη είναι μια τυχαία διαδικασία, για το λόγο αυτό έχουμε ορίσει μια πιθανότητα μέσα στα όρια της οποίας μπορεί να συμβεί μετάλλαξη.

Αφού ολοκληρώσαμε τη διαδικασία της αναπαραγωγής υπολογίζουμε την ικανότητα των παιδιών και αντικαθιστούμε τα άτομα του πληθυσμού με τη μεγαλύτερη ικανότητα (υπενθυμίζουμε εδώ ότι το βάρος της διαδρομής ισούται με την ικανότητα). Τέλος αξίζει να σημειωθεί ότι ο αλγόριθμος εκτελείται για έναν προκαθορισμένο αριθμό επαναλήψεων.

ΚΕΦΑΛΑΙΟ 3

Η τεχνολογία Stretch και Σχετικές Μέθοδοι Βελτιστοποίησης

Στο κεφάλαιο αυτό θα προσπαθήσουμε να δώσουμε μερικές πληροφορίες για το εργαλείο της Stretch το οποίο και χρησιμοποιήσαμε για την υλοποίηση αυτής της εργασίας καθώς και για τις μεθόδους που χρησιμοποιούνται για να επιτύχουμε τη βελτιστοποίηση αλγορίθμων.

3.1 Στόχος της Stretch

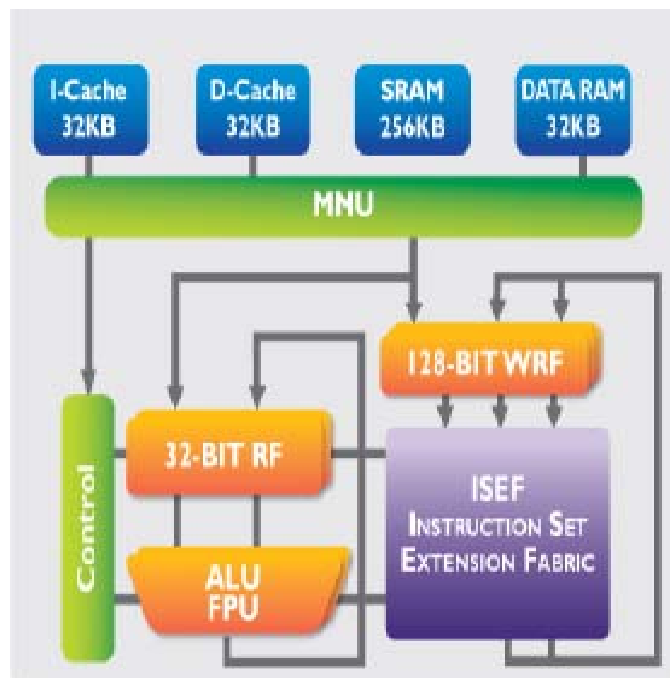
Όλες οι βιομηχανίες που ασχολούνται με την κατασκευή επεξεργαστών προσπαθούν να συνδυάσουν επεξεργαστές γενικού σκοπού με FPGAs ώστε να επιτύχουν μέγιστη απόδοση σε σύντομο χρόνο. Η Stretch είναι μια νέα τεχνολογία σχεδίασης επεξεργαστών η οποία καταφέρνει να συνδυάσει τους δύο αυτούς παράγοντες. Πιο συγκεκριμένα προσφέρει έναν τρόπο ώστε να ενσωματώσουμε την προγραμματιστική λογική μέσα στον επεξεργαστή και να προγραμματίσουμε τη δομή αυτή χρησιμοποιώντας ένα συμβατικό C/C++ περιβάλλον.

Έχοντας ως πρόκληση την επιτυχία της μέγιστης απόδοσης η εταιρεία Stretch είναι η πρώτη που δημιούργησε μια αρχιτεκτονική η οποία επιτυγχάνει τα ακόλουθα:

- Καλά αποτελέσματα χρησιμοποιώντας ταυτόχρονα λύσεις υλικού και λογισμικού.
- Μειώνει το κόστος του συστήματος.
- Απευθύνεται σε οποιαδήποτε εφαρμογή περιέχει τμήματα κώδικα τα οποία απαιτούν μεγάλη υπολογιστική ισχύ (compute-intensive εφαρμογές).

3.2 Η αρχιτεκτονική της Stretch

Η εταιρεία Stretch κατασκεύασε τη σειρά ρυθμιζόμενων από λογισμικό επεξεργαστών S5000, η οποία είναι βασισμένη στον πυρήνα Tensilica RISC επεξεργαστή και ταυτόχρονα ενσωματώνει ένα μικρό αναδιατασσόμενο τμήμα, όπως φαίνεται στο *Σχήμα 3.1*.



Σχήμα 3.1

Οι επεξεργαστές S5000 ενσωματώνουν τον Tensilic Xtensa RISC πυρήνα επεξεργαστή και το αναδιατασσόμενο κομμάτι ISEF (Instruction Set Extension Fabric). Το ISEF είναι μια προγραμματιστική λογική για ενσωματωμένες μονάδες με το οποίο μπορούμε να διατυπώσουμε τα compute intensive μέρη της εφαρμογής. Η οικογένεια επεξεργαστών S5000 παρέχει δύο ανεξάρτητες μονάδες ISEF, ISEF A και ISEF B , οι οποίες μπορούν να αρχικοποιηθούν και να χρησιμοποιηθούν ανεξάρτητα η μία από την άλλη. Για να προγραμματίσουμε τους επεξεργαστές S5000 χρησιμοποιούμε ως γλώσσα προγραμματισμού C/C++. Η Stretch C είναι μια γλώσσα παραπλήσια με τη C η οποία περιέχει επεκτάσεις ώστε να είναι εφικτή η υλοποίηση της εφαρμογής σε υλικό και χρησιμοποιείται για την παρουσίαση των κρίσιμων σημείων της σχεδίασης στα αναδιατασσόμενα κομμάτια του επεξεργαστή.

Το αναδιατασσόμενο κομμάτι του επεξεργαστή περιέχει ένα ολοκληρωμένο σύστημα πόρων για υπολογισμούς. Πιο συγκεκριμένα υπάρχουν δύο ανεξάρτητα σετ υπολογιστικών πόρων: ένα για αριθμητικές και λογικές πράξεις (AU) και ένα για πολλαπλασιασμούς και ολισθήσεις (MU). Ο αριθμός των AU και MU για κάθε αναδιατασσόμενο κομμάτι είναι 4096 και 8192 αντίστοιχα. Το άθροισμα των υπολογιστικών μονάδων που χρησιμοποιούνται από όλες τις εντολές σε κάθε ISEF δε μπορεί να υπερβαίνει το συνολικό αριθμό των υπολογιστικών μονάδων κάθε ISEF.

Τελειώνοντας πρέπει να αναφέρουμε τα πλεονεκτήματα της οικογένειας επεξεργαστών S5000:

- Ενισχύει την απόδοση του συστήματος για compute-intensive εφαρμογές.
- Επιτρέπει τη γρήγορη ανάπτυξη του συστήματος από τη στιγμή που ξεκινάει η παραγωγή του μέχρι να βγει στην αγορά.
- Μειώνει το κόστος ανάπτυξης του συστήματος.
- Παρέχει γρήγορη επικοινωνία του επεξεργαστή με τον έξω κόσμο.
- Απαλάσσει τους μηχανικούς ενσωματωμένων συστημάτων από χρήση γλωσσών όπως η Verilog και η VHDL και τις αντικαθιστά με γλώσσες υψηλού επιπέδου όπως η C και η C++.

3.3 Το περιβάλλον ανάπτυξης της Stretch

Με το περιβάλλον ανάπτυξης της Stretch (Stretch Integrated Development Enviroment, Stretch IDE) οι σχεδιαστές συστημάτων χρησιμοποιούν απλές μεθόδους C/C++ για να δημιουργήσουν ολοκληρωμένες εφαρμογές οι οποίες περίεχουν τα ειδικά χαρακτηριστικά των προγραμματιζόμενων επεξεργαστών.

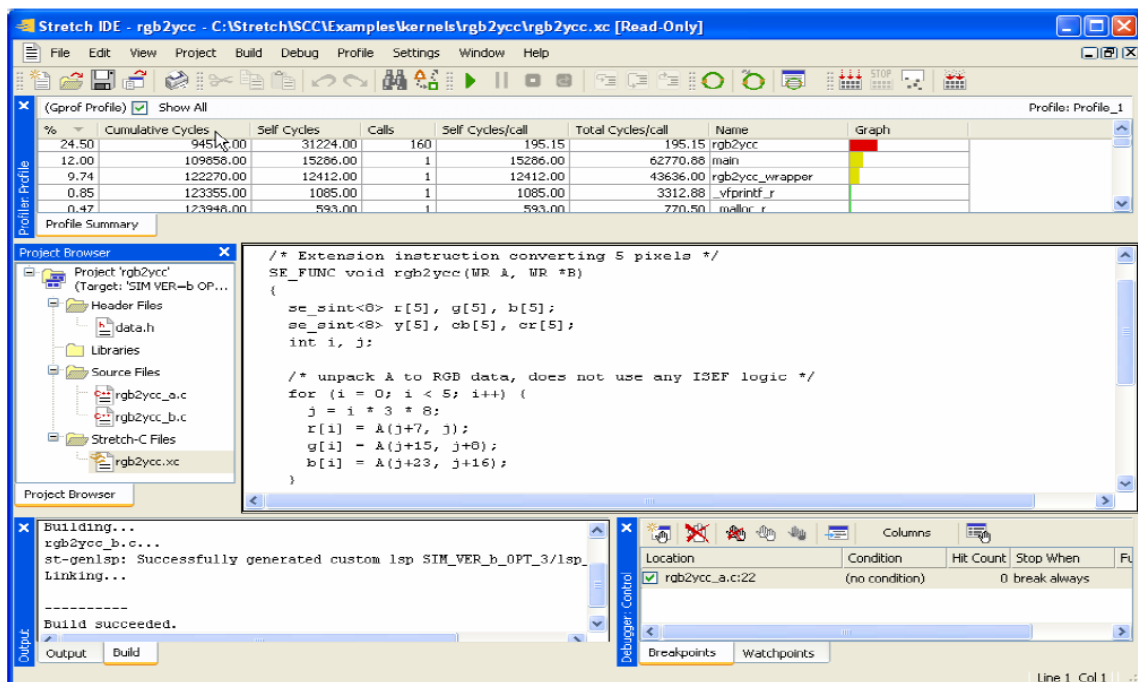
Αρχικά, το Stretch IDE αποτελείται από έναν compiler ο οποίος μας δίνει τη δυνατότητα να χρησιμοποιήσουμε C/C++ ώστε να προγραμματίσουμε τον επεξεργαστή μας και το αναδιατασσόμενο κομμάτι του. Ταύτοχρονα έχουμε χρήσιμα χαρακτηριστικά για debugging όπως breakpoints και δυνατότητα παρακολούθησης της τιμής μια συγκεκριμένης μεταβλητής.

Στη συνέχεια, μας παρέχει την δυνατότητα μέσω της επιλογής Profiling να εντοπίσουμε τα σημεία του προγράμματός μας στα οποία

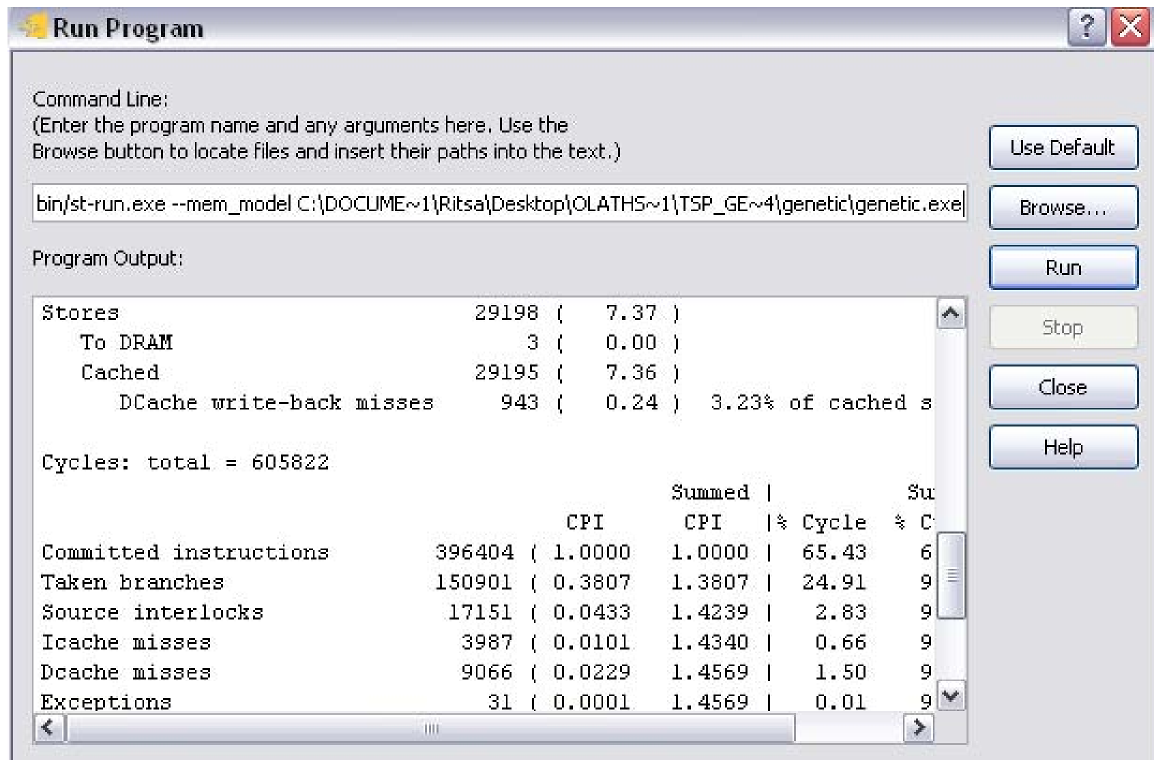
καταναλώνεται ο περισσότερος χρόνος και είναι υποψήφια για βελτιστοποίηση. Ο profiler συγκεντρώνει τα στοιχεία από την πραγματική εκτέλεση του προγράμματος.

Τέλος, όταν εκτελέσουμε το πρόγραμμα, με χρήση της εντολής run, εμφανίζεται ένα παράθυρο (Program Output) το οποίο εκτός από την έξοδό του προγράμματος μας επιτρέπει να πάρουμε χρήσιμες πληροφορίες για τα χαρακτηριστικά εκτέλεσης του, όπως για παράδειγμα τους συνολικούς κύκλους ρολογιού και τα data cache misses. Πρέπει να σημειώσουμε ότι όταν κάνουμε χρήση του STDOUT τα αποτελέσματα μας θα εμφανίζονται στο παράθυρο Program Output.

Τα σχήματα 3.2 και 3.3 μας δίνουν μια γενική εικόνα του Stretch IDE και των αποτελεσμάτων που παίρνουμε κατά την εκτέλεση του προγράμματος.



Σχήμα 3.2



Σχήμα 3.3

3.4 Βελτιστοποίηση με τους επεξεργαστές της Stretch

Πριν ξεκινήσουμε να αναλύουμε τον τρόπο με τον οποίο επιτύχαμε τη βελτιστοποίηση των αλγορίθμων καλό θα ήταν να δώσουμε μια γενική ιδέα για τις μεθόδους που χρησιμοποιήσαμε προκειμένου να φτάσουμε στο επιθυμητό αποτέλεσμα.

3.4.1 Εύρεση compute intensive τμημάτων κώδικα

Το πρώτο βήμα που θα πρέπει να ακολουθούμε για την βελτιστοποίηση ενός κώδικα είναι ο προσδιορισμός των τμημάτων του, τα οποία καταναλώνουν μεγάλο ποσοστό κύκλων ως προς το συνολικό αριθμό κύκλων του κώδικα. Ο προσδιορισμός αυτός γίνεται με δύο κυρίως τρόπους:

1. Μέσω της διαδικασίας profiling
2. Μέσω μιας συνάρτησης που επιστρέφει τον αριθμό των κύκλων που έχουν περάσει από την αρχή της εκτέλεσης.

Το πρόβλημα με την διαδικασία του profiling είναι ότι, μπορεί και υπολογίζει τον αριθμό των κύκλων σε κάθε συνάρτηση ξεχωριστά και όχι συγκεκριμένα σε ένα κομμάτι κώδικα. Η συγκεκριμένη διαδικασία γίνεται με τη βοήθεια της συνάρτησης **sx_get_ccount()**, η οποία επιστρέφει τον συνολικό αριθμό κύκλων από την αρχή της εκτέλεσης του προγράμματος μέχρι εκείνο το σημείο που θα καλεστεί. Εδώ πρέπει να αναφέρουμε ότι για να μπορέσουμε να χρησιμοποιήσουμε τη συνάρτηση sx_get_ccount() θα πρέπει να εισάγουμε στον κώδικά μας τη βιβλιοθήκη sx-timer.h.

3.4.2 Βελτιστοποίηση κώδικα

Για την βελτιστοποίηση του κώδικα στους επεξεργαστές S5000 ακολουθούνται συγκεκριμένα βήματα.

3.4.2.1 Χρήση "προσαρμοζόμενων" εντολών

Εφόσον έχουμε εντοπίσει τα σημεία του κώδικά μας που καταναλώνουν των περισσότερο χρόνο κάνουμε χρήση **"προσαρμοζόμενων" εντολών**. Στο βήμα αυτό το τμήμα του κώδικα με τους περισσότερους κύκλους τοποθετείται στο αναδιατασσόμενο κομμάτι του επεξεργαστή S5000 και υλοποιείται σαν μία εντολή. Για να το επιτύχουμε αυτό δημιουργούμε στο project μας ένα νέο αρχείο με κατάληξη .xc το οποίο περιέχει κώδικα

για το τμήμα της αναδιατασσόμενης λογικής και κάνουμε include τη βιβλιοθήκη stretch.h. Οι συναρτήσεις που μπαίνουν σε αναδιατασσόμενη λογική πρέπει να δηλώνονται SE_FUNC void. Εφόσον η συνάρτηση που αποτελεί το αναδιατασσόμενο κομμάτι είναι void τα αποτελέσματά της επιστρέφονται με αναφορά.

Για να εισάγουμε τις τιμές των δεδομένων στο αναδιατασσόμενο κομμάτι καθώς και για να επιστρέψουμε τα αποτελέσματα χρησιμοποιούμε 128-bit wide registers (WR). Το αρχείο καταχωρητών (register file) αποτελείται από δύο μέρη, A και B, κάθε ένα από τα οποία συνίσταται από 16 wide registers (το WR αρχείο μπορεί να αντιμετωπιστεί σαν δύο διαφορετικά αρχεία καταχωρητών WRA και WRB). Σε κάθε περίπτωση όμως, λόγω του περιορισμένου μεγέθους του αναδιατασσόμενου κομματιού του επεξεργαστή, μπορούμε να έχουμε ταυτόχρονα τρείς WR για είσοδο και δύο για έξοδο.

Υπάρχουν πολλές εντολές φόρτωσης και αποθηκεύσης για το αρχείο των wide register ανάλογα με τον τύπο των δεδομένων που θέλουμε να μεταφέρουμε από και προς το αναδιατασσόμενο κομμάτι. Στη συγκεκριμένη εργασία τα δεδομένα μας είναι ακέραιοι οπότε χρησιμοποιήσαμε τις εντολές που φαίνονται στο Σχήμα 3.4, όπου το νούμερο σε κάθε εντολή δηλώνει τον αριθμό των bits του καταχωρητή που δεσμεύουμε κάθε φορά.

load		**store**	
into WRA	into WRB	from WRA	from WRB
WRAL8X	WRBL8X	WRAS8X	WRBS8X
WRAL16X	WRBL16X	WRAS16X	WRBS16X
WRAL32X	WRBL32X	WRAS32X	WRBS32X
WRAL64X	WRBL64X	WRAS64X	WRBS64X
WRAL128X	WRBL128X	WRAS128X	WRBS128X

Σχήμα 3.4

3.4.2.2 Εκτέλεση Υπολογισμών σε Παραλληλισμό

Το δεύτερο βήμα για τη βελτιστοποίηση του κώδικα είναι η εκτέλεση εντολών παράλληλα εκμεταλλευόμενοι τα 128 bits των wide registers του αναδιατασσόμενου κομματιού. Πιο συγκεκριμένα φορτώνουμε σε κάθε wide register διαδοχικές τιμές της μεταβλητής (στην περίπτωση αυτή συνήθως μιλάμε για διατάξεις) και τις επεξεργαζόμαστε ταυτόχρονα. Το βήμα αυτό δεν είναι εφικτό να πραγματοποιηθεί σε όλους τους κώδικες.

3.4.2.3 Manual Loop Unrolling

Η μέθοδος manual loop unrolling (ξεδίπλωμα βρόγχου) χρησιμοποιείται όταν το τμήμα του κώδικα που θέλουμε να βελτιστοποιήσουμε αποτελεί τμήμα ενός βρόχου. Παρόλο που έχουμε ήδη ένα ποσοστό βελτιστοποίησης με τις προηγούμενες μεθόδους ως προς τον αριθμό των κύκλων που παίρνει η εκτέλεση του βρόχου η μέθοδος αυτή παρέχει περαιτέρω βελτιστοποίηση.

Γενικά όταν ένας βρόχος επαναλαμβάνετε πολλές φορές, αυτό έχει ως αποτέλεσμα να γίνεται πολλές φορές η εκτέλεση της εντολής jump, στο τέλος του βρόχου. Η εκτέλεση της εντολής αυτής οδηγεί στο σβήσιμο της ομοχειρίας (pipeline) του επεξεργαστή με αποτέλεσμα να οδηγούμαστε σε χάσιμο κάποιων κύκλων. Αυτό που μπορεί να γίνει ώστε να μειώσουμε τον αριθμό των χαμένων κύκλων λόγω ανασχέσεων (stalls) στην ομοχειρία είναι το manual unrolling του βρόχου.

3.4.2.4 Χρήση μνήμης υψηλών επιδόσεων

Μια άλλη αιτία η οποία μπορεί να αυξάνει τους κύκλους της εκτέλεσης ενός τμήματος κώδικα είναι τα data cache misses (αστοχίες μνήμης). Όταν τα δεδομένα στα οποία γίνεται πρόσβαση, βρίσκονται στη μνήμη cache του επεξεργαστή τότε δεν υπάρχουν data cache misses με αποτέλεσμα να μην έχουμε χαμένους κύκλους. Μια από τις δυνατότητες της τεχνολογίας των επεξεργαστών της Stretch είναι ότι ο χρήστης μπορεί να τοποθετεί τις διατάξεις εισόδου και εξόδου στην cache του επεξεργαστή μειώνοντας κατα συνέπεια τα data cache misses. Αυτό γίνεται με την προσθήκη του __attribute__ κατά τη δήλωση των μεταβλητών που επιθυμεί ο χρήστης να εισάγει στην εσωτερική μνήμη cache του επεξεργαστή.

3.4.2.5 Περαιτέρω βελτιστοποιήσεις

Εκτός από τη χρήση μνήμης υψηλών επιδόσεων μπορούμε να "γλιτώσουμε" data cache misses ορίζοντας ως static της μεταβλητές του

κώδικά μας όπου αυτό είναι εφικτό. Με τον τρόπο αυτό οι μεταβλητές βρίσκονται σε σταθερές θέσεις μνήμης και προσπελαύνονται ευκολότερα.

Το τελευταίο βήμα για τη βελτιστοποίηση ενός κώδικα, που μπορεί να εφαρμοστεί είναι η επιλογή Optimization level που μας παρέχει η Stretch. Η επιλογή αυτή βρίσκεται στο Project→ Project Properties, παίρνει τιμές O0, O1, O2, O3 και καθορίζει στον compiler το μέγεθος της βελτιστοποίησης που θα εφαρμόσει στον κώδικα. Προφανώς, όσο μεγαλύτερη έιναι η επιλογή του Optimization τόσο περισσότερο χρόνο διαρκεί το compilation το κώδικα. Η βελτιστοποίηση που προσφέρει η επιλογή αυτή είναι, κυριώς το loop unrolling, το οποίο δεν είναι πάντοτε επιθυμητό, αφού οδηγεί σε μεγάλη αύξηση των γραμμών του κώδικα.

ΚΕΦΑΛΑΙΟ 4

Βελτιστοποίηση του αλγόριθμου TSP

Στο κεφάλαιο αυτό όπως και στα επομένα δύο κεφάλαια περιγράφουμε τα βήματα που ακολουθήσαμε για να επιτύχουμε τη βελτιστοποιήση των αλγορίθμων που μελετήσαμε στο Κεφάλαιο 2 με τα εργαλεία της Stretch, τα αποτελέσματα που πήραμε, καθώς και τα προβλήματα που αντιμετωπίσαμε στην προσπάθεια αυτή. Στο παρόν κεφάλαιο ασχολούμαστε με τον αλγόριθμο TSP ενώ στα κεφάλαια 5 και 6 με το MINISAT και τον γενετικό αλγόριθμο αντίστοιχα. Σε κάθε κεφάλαιο αρχικά δίνουμε μερικά στοιχεία για τον κώδικα και στη συνέχεια περιγράφουμε τη διαδικασία βελτιστοποίησης.

4.1 Υλοποίηση TSP

Όπως αναφέραμε στην ενότητα 2.1.3 οι αλγόριθμοι που χρησιμοποιήθηκαν για την επίλυση του προβλήματος του πλανόδιου πωλητή είναι ο *furthest inserstion* για την κατασκευή της διαδρομής και ο *2-opt* για τη βελτιστοποίηση της αρχικής διαδρομής. Πριν ξεκινήσουμε την ανάλυση της υλοποίησης πρέπει να αναφέρουμε ότι η γλώσσα που χρησιμοποιήθηκε σε όλους τους κώδικες στην παρούσα εργασία είναι η

C και κατ’ επέκταση η Stretch C και τα εργαλεία που χρησιμοποιήσαμε ήταν το Microsoft Visual Studio 6 και το Stretch IDE. Επίσης στο σημείο αυτό αξίζει να πούμε ότι δοκιμάσαμε τον αλγόριθμο TSP για 10 πόλεις. Ο λόγος που επιλέξαμε αυτό τον αριθμό είναι για να μπορούμε να πάρουμε τα αποτελέσματά του σε ικανοποιητικό χρόνο.

Για να επιλύσουμε τον TSP με χρήση των αλγορίθμων που αναφέραμε αρχικά, σκεφτήκαμε ότι ο ευκολότερος τρόπος για την εισαγωγή των δεδομένων θα ήταν η χρήση ενός αρχείου το οποίο θα διαβάζουμε μέσα από το πρόγραμμα και θα περιέχει τη βιβλιοθήκη TSPLIB καθώς και τον αριθμό των πολέων και μια ακόμα τιμή η οποία θα είναι μεγαλύτερη από οποιαδήποτε τιμή της TSPLIB. Η ανάγνωση όμως από αρχείο απαιτεί πολλούς κύκλους και έτσι θεωρήσαμε σωστότερη τη δημιουργία ενός header file που να περιέχει αυτά τα δεδομένα (ο αριθμός των κύκλων για τις δύο αυτές περιπτώσεις παρουσιάζεται στον Πίνακα 4.1 στο τέλος της ενότητας 4.2). Έτσι τελικά δημιουργήσαμε τέσσερα αρχεία, τα οποία αναφέρονται παρακάτω:

- data.h : το αρχείο αυτό περιέχει τη μεταβλητή n η οποία ορίζει τον αριθμό των πόλεων, έναν πίνακα w[100] ο οποίος περιέχει τις τιμές της βιβλιοθήκης TSPLIB, σε μορφή γραμμής και ορίζεται ως static γιατί αρχικοποιείται μια φορά και δεν αλλάζει κατά τη διάρκεια της εκτέλεσης και τέλος τη μεταβλητή inf η οποία ορίζει έναν ακέραιο μεγαλύτερο από το μέγιστο στοιχείο του πίνακα w. Όπως είναι φανερό αν κάποιος θέλει να δοκιμασεί διαφορετικό αριθμό πόλεων ή διαφορετική βιβλιοθήκη TSPLIB αρκεί να αλλάξει τα δεδομένα του συγκεκριμένου αρχείου.

- fitsp.h : το αρχείο αυτό περιέχει τη συνάρτηση fitsp() η οποία παίρνει ως ορίσματα τον αριθμό των πόλεων n, την τιμή inf, τον πίνακα w, τον πίνακα route[n] στον οποίο αποθηκεύουμε τη διαδρομή και τη μεταβλητή tweight η οποία μας δίνει το συνολικό κόστος της διαδρομής. Μέσα στη συνάρτηση fitsp() υλοποιούμε τον αλγόριθμο *furthest insertion* με τον τρόπο που περιγράφηκε στην ενότητα 2.1.3.
- twoopt.h : το αρχείο αυτό περιέχει τη συνάρτηση twoopt() η οποία υλοποιεί τον αλγόριθμο *2-opt* και παίρνει τα ίδια ορίσματα με τη συνάρτηση fitsp() εκτός της μεταβλητής inf.
- main.c : το κεντρικό αρχείο στο οποίο περιέχεται η συνάρτηση main() μέσα στην οποία γίνεται η κλήση των συναρτήσεων fitsp() και twoopt() και η εκτύπωση των αποτελεσμάτων.

4.2 TSP και Stretch

Ξεκινώντας την προσπάθεια της βελτιστοποίησης μας τρέξαμε το πρόγραμμά μας με την εντολή run και κάναμε profiling ώστε να δούμε το συνολικό αριθμό κύκλων του προγράμματος αλλά και την κατανομή των κύκλων στις συναρτήσεις μας. Έτσι οι συνολικοί κύκλοι του προγράμματος μας είναι 82553, τα data cache misses είναι 1999 και τα αποτελέσματα του profiling παρουσιάζονται στο Σχήμα 4.2.

(Gprof Profile) Show All

%	Cumulative Cycles	Self Cycles	Calls	Self Cycles/call	Total Cycles/call	Name	Graph
24.89	20548.00	20548.00	??	??	??	ResetH	
15.72	33522.00	12974.00	1	12974.00	12974.00	twoopt	
14.69	45649.00	12127.00	24	505.29	1109.07	_vfprintf_r	
14.07	57263.00	11614.00	1	11614.00	12518.83	fitsp	
5.55	61847.00	4584.00	46	99.65	175.08	__sfvwrite	
4.09	65225.00	3378.00	1	3378.00	3378.00	xthal_dcache_all_writeback	
2.56	67335.00	2110.00	??	??	??	do_pte	
2.03	69007.00	1672.00	50	33.44	33.44	memmove	
1.99	70649.00	1642.00	122	13.46	13.46	_mbtowc_r	
1.48	71868.00	1219.00	50	24.38	24.38	memchr	
1.09	72767.00	899.00	4	224.75	268.92	_malloc_r	
1.03	74464.00	848.00	46	18.43	193.51	__sprint	
1.03	73616.00	849.00	24	35.38	35.38	memset	
1.00	75291.00	827.00	24	34.46	1181.90	printf	

Σχήμα 4.2

Όπως φαίνεται από το Σχήμα 4.2 οι περισσότεροι κύκλοι καταναλώνονται στη συνάρτηση twoopt() (βλέπουμε ότι η συνάρτηση fitsp() καταναλώνει αντίστοιχο αριθμό κύκλων αλλά στη συγκεκριμένη εργασία μας ενδιέφερε να δούμε αν και κατά πόσο βελτιστοποιείται η συνάρτηση twoopt()). Η συνάρτηση αυτή αποτελείται από δύο βρόχους for και ένα βρόχο do…while. Εφαρμόζοντας την συνάρτηση sx_get_ccount() διαπιστώσαμε ότι ο βρόχος do…while παίρνει τους περισσότερους κύκλους, για την ακριβεία παίρνει 12181 κύκλους. Συνεπώς αυτό είναι το κομμάτι που πρέπει να βελτιστοποίησουμε.

Ο βρόχος do…while περιέχει στο εσωτερικό του δύο βρόχους for και έναν ακόμα do…while καθώς επίσης και αναθέσεις. Με μια πρώτη ματιά μπορούμε να καταλάβουμε ότι ουσιαστικά η υλοποίηση πολλαπλών αναθέσεων στο αναδιατασσόμενο είναι ανέφικτη λόγω του περιορισμένου μεγέθους του και ταυτόχρονα μπορεί να οδηγήσει σε μεγαλύτερη κατανάλωση κύκλων λόγω των εντολών μετάβασης από και

προς το αναδιατασσόμενο. Για το λόγο αυτό στο αναδιατασσόμενο επιλέξαμε να υπολογίζουμε, από τους δύο βρόχους for, τις συνθήκες των εντολών if καθώς και αριθμητικές πράξεις, που καταναλώνουν πολλούς κύκλους, όπως είναι ο υπολογισμός του βάρους μεταξύ διαδοχικών κόμβων τις διαδρομής.

Το επόμενο βήμα για τη βελτιστοποίηση του κώδικα που ήταν δυνατό να εφαρμοστεί στη συγκεκριμένη περίπτωση ήταν η χρήση μνήμης υψηλών επιδόσεων. Πιο συγκεκριμένα τοποθετήσαμε τον πίνακα w στη cache του επεξεργαστή και παρατηρήσαμε μείωση τόσο στα data cache misses όσο και στους συνολικούς κύκλους εκτέλεσης.

Το τελευταίο βήμα είναι η αλλαγή του optimization level σε O3 με χρήση του οποιού ο compiler έκανε τις τελικές βελτιστοποιήσεις του κώδικα.
Όλα τα αποτελέσματα που πήραμε από τα διαδοχικά βήματα βελτιστοποίησης που εφαρμόσαμε φαίνονται στον Πίνακα 4.1 που ακολουθεί. Ο πίνακας αυτός μας δίνει τους συνολικούς κύκλους τα data cache misses και το ποσοστό βελτιστοποίησης για κάθε βήμα ξεχωριστά. Σημειώνουμε ότι το πρώτο ποσοστό βελτιστοποίησης είναι μεταξύ του προγράμματος με εισαγωγή των δεδομένων από αρχείο και από header file, ενώ τα υπόλοιπα είναι μεταξύ του προγράμματος με εισαγωγή δεδομένων από header file και καθ’ενός από τα βήματα βελτιστοποίησης.

Βήμα βελτιστοποίησης	Συνολικοί Κύκλοι	Data Cache Misses	Ποσοστό Βελτιστοποίησης(%)
Αρχικά με εισαγωγή των δεδομένων από αρχείο	172668	2839	--
Αρχικά με εισαγωγή των δεδομένων από header file	82553	1999	52.1
Βελτιστοποίηση 1ου for-loop	81252	1989	1.5
Βελτιστοποίηση 2ου for-loop	82080	1979	0.57
Βελτιστοποίηση και των δύο for-loop ταυτόχρονα	81205	1984	1.63
Χρήση μνήμης υψηλών επιδόσεων	81047	1737	1.82
Βελτιστοποίηση του compiler	65805	1741	20.29

Πίνακας 4.1

Μετά από αυτή τη βελτιστοποίηση βλέπουμε ότι ο κώδικάς μας εκτελείται σε 219,35 μsec εφόσον το ρολόι του επεξεργαστή της Stretch είναι 300MHz. Στο παράρτημα εμφανίζονται τα βελτιστοποιημένα τμήματα του κώδικα καθώς και το τμήμα του αναδιατασσόμενου.

4.3 Συμπεράσματα

Όπως είδαμε στην προηγούμενη ενότητα μετά από την προσπάθεια βελτιστοποίησης καταφέραμε να κάνουμε τον κώδικά μας να εκτελείται

περίπου 2 φορές γρηγορότερα από ότι ο αρχικός κώδικας που ήταν γραμμένος σε απλή C. Η βελτιστοποίηση αυτή θα μπορούσε να είναι μεγαλύτερη αν είχαμε τη δυνατότητα να περάσουμε στο αναδιατασσόμενο κομμάτι μεγαλύτερο αριθμό δεδομένων έτσι ώστε μέσα σε αυτό να εκτελούμε περισσότερες πράξεις. Επίσης η απλή δομή του κώδικα στη συγκεκριμένη περίπτωση στάθηκε εμπόδιο στη βελτιστοποίηση γιατί όπως αναφέραμε και προηγουμένως αν τοποθετήσουμε έναν αριθμό αναθέσεων στο αναδιατασσόμενο η μετάβαση από και προς το αναδιατασσόμενο καταναλώνει περισσότερο χρόνο από ότι αν οι εντολές αυτές εκτελούνται κανονικά μέσα στον κώδικα. Ένας τελευταίος παράγοντας που οδήγησε σε αυτό το αποτέλεσμα είναι ότι τελικά για να υλοποιηθεί ο αλγόριθμος 2-opt αυτό που ουσιαστικά κάναμε από πλευράς κώδικα είναι ένας μεγάλος βρόχος ο οποίος μέσα του περιέχει μικρότερους και οι οποίοι με τη σειρά τους περιλαμβάνουν απλές εντολές η τοποθέτηση των οποίων στο αναδιατασσόμενο έχει ως αποτέλεσμα την αύξηση των κύκλων και όχι τη μείωσή τους.

ΚΕΦΑΛΑΙΟ 5

Βελτιστοποίηση του MINISAT

Ακολουθώντας την ίδια διαδικασία με τον TSP στο κεφάλαιο αυτό δίνουμε λεπτομέρειες για την υλοποίηση και τη βελτιστοποίηση του MINISAT.

5.1 Υλοποίηση MINISAT

Πριν ξεκινήσουμε την ανάλυση την υλοποίησης και αργότερα της βελτιστοποίησης πρέπει να σημειώσουμε ότι ο κώδικας του MINISAT είναι αυτός που μπορεί να βρει κανείς στην ιστοσελίδα www.minisat.se και είναι ο πρωτότυπος κώδικας του MINISAT σε C. Για το λόγο αυτό και επειδή οι δημιουργοί του φρόντισαν ώστε από προγραμματιστικής άποψης να είναι όσο το δυνατόν βέλτιστος δεν τον τροποποιήσαμε για τις ανάγκες τις εργασίας αλλά προσπαθήσαμε να βελτιστοποιήσουμε το υπάρχον υλικό. Επίσης πρέπει να αναφέρουμε ότι και πάλι για λόγους χρόνου αποφασίσαμε να τρέξουμε τον κώδικα για 500 μεταβλητές, 50 clauses και το πολύ μέχρι 5 literals σε κάθε clause.

Ο MINISAT παίρνει ως είσοδο ένα αρχείο input.txt η μορφή του οποίου είναι:

+1 -4 0
-3 -7 +9 +5 -1 0
+2 +8 -4 0
+6 +3 +5 +9 0

Κάθε γραμμή του αρχείου αποτελεί ένα clause και το 0 στο τέλος της κάθε γραμμής σηματοδοτεί το τέλος του clause. Με + συμβολίζουμε τη μεταβλητή και με – το συμπλήρωμά της. Τέλος το κάθε νούμερο συμβολίζει τη μεταβλητή στην οποία αναφερόμαστε κάθε φορά. Έτσι για το συγκεκριμένο αρχείο εισόδου ο solver ελέγχει αν ικανοποιείται ή όχι η πρόταση (το συμπλήρωμα απεικονίζεται με το σύμβολο ¬):

(x_1 OR $\neg x_4$) AND ($\neg x_3$ OR $\neg x_7$ OR x_9 OR x_5 OR $\neg x_1$) AND (x_2 OR x_8 OR $\neg x_4$) AND (x_6 OR x_3 OR x_5 OR x_9).

Συνεχίζοντας την περιγραφή του MINISAT θα δούμε τα αρχεία που αποτελούν το πρόγραμμά μας. Τα αρχεία αυτά είναι:

- solver.h : το αρχείο αυτό περιέχει τους ορισμούς των δομών solver και clause.
- vec.h : στο αρχείο αυτό ορίζονται δύο δομές veci_t και vecp_t οι οποίες αποτελούν ουσιαστικά τους ορισμούς διατάξεων ακεραίων και δεικτών αντίστοιχα οι οποίοι χρησιμοποιούνται για να κρατούν τα διάφορα στοιχεία του προγράμματος.
- main.c : το κύριο αρχείο του προγράμματος στο οποίο γίνεται η ανάγνωση του αρχείου η δημιουργία του solver και η επιστροφή των αποτελεσμάτων του.
- solver.c : στο αρχείο αυτό υλοποιούνται όλα τα χαρακτηριστικά του solver όπως αυτά αναφέρθηκαν στην ενότητα 2.2.3.

5.2 MINISAT και Stretch

Όπως και προηγουμένως το πρώτο βήμα για τη βελτιστοποίηση του κώδικα είναι το profiling, τα αποτελέσματα του οποίου παρουσιάζουμε στο Σχήμα 4.3.

(Gprof Profile) Show All

%	Cumulative Cycles (K)	Self Cycles (K)	Calls	Self Cycles/call (K)	Total Cycles/call (K)	Name	Graph
21.05	545.89	545.89	1010	0.54	0.54	__divdf3	
5.78	695.91	150.02	1067	0.14	0.14	__muldf3	
4.91	823.31	127.40	488	0.26	1.98	order_select	
4.85	949.07	125.76	998	0.13	0.16	order_update	
4.58	1067.74	118.67	1092	0.11	0.12	_free_r	
4.35	1180.58	112.83	490	0.23	0.37	solver_propagate	
4.30	1292.18	111.61	1984	0.06	0.06	veci_push	
3.75	1389.39	97.20	1096	0.09	0.10	_malloc_r	
3.26	1474.02	84.63	1	84.63	1600.58	solver_search	
3.20	1557.07	83.05	50	1.66	7.12	solver_setnvars	
3.04	1635.82	78.75	505	0.16	0.23	enqueue	
1.92	1685.60	49.78	226	0.22	0.30	parseInt	
1.69	1729.43	43.83	1000	0.04	0.15	vecp_new	
1.67	1772.77	43.33	497	0.09	1.55	drand	
1.62	1814.69	41.93	2990	0.01	0.01	veci_size	
1.60	1856.19	41.50	487	0.09	0.39	assume	
1.46	1894.03	37.83	50	0.76	9.00	solver_addclause	
1.42	1930.96	36.93	2636	0.01	0.01	lit_var	
1.40	1967.33	36.37	499	0.07	0.31	order_unassigned	
1.32	2001.62	34.29	1	34.29	192.63	solver_delete	
1.26	2034.36	32.74	531	0.06	0.06	__subdf3	
1.18	2065.07	30.72	2	15.36	99.02	solver_canceluntil	
1.16	2095.26	30.19	16	1.89	6.18	_vfprintf_r	
1.11	2124.16	28.90	1517	0.02	0.02	__ltdf2	

Σχήμα 4.3

Εξετάζοντας τα αποτελέσματα που πήραμε από το profiling του κώδικα παρατηρούμε ότι οι περισσότεροι κύκλοι καταναλώνονται σε πράξεις όπως η διαίρεση και στη συνέχεια ο πολλαπλασιασμός. Οι πράξεις αυτές είναι διασκορπισμένες στον κώδικα πράγμα το οποίο καθιστά αδύνατη την είσοδό τους στο αναδιατασσόμενο κομμάτι. Στη συνέχεια ακολουθεί η σύναρτηση order_select(), η οποία ουσιαστικά επιστρέφει τις μεταβλητές με το μεγαλύτερο activity, που ήταν και η πρώτη συνάρτηση που προσπαθήσαμε να βελτιστοποιήσουμε. Η δομή της συνάρτησης

order_select() ήταν το μεγαλύτερο πρόβλημα στην προσπάθεια βελτιστοποίησης. Η συνάρτηση ξεκινάει με κλήσεις άλλων συναρτήσεων, ακολουθούν αναθέσεις με χρήση δεικτών και στη συνέχεια ένας βρόχος while ο οποίος περιέχει νέες κλήσεις συναρτήσεων και ένα δεύτερο βρόχο while μέσα στον οποίο έχουμε δύο συνθήκες if, μια κλήση συνάρτησης και αναθέσεις. Όπως είναι προφανές οι κλήσεις συναρτήσεων δε μπορούν να μπουν στο αναδιατασσόμενο κομμάτι. Επίσης το αναδιατασσόμενο κομμάτι δεν υποστηρίζει τη χρήση δεικτών. Κατά συνέπεια το τμήμα του κώδικα που προσπαθήσαμε να βάλουμε μέσα στο αναδιατασσόμενο είναι ο δεύτερος βρόχος while και συγκεκριμένα οι αναθέσεις και η πράξη που γίνεται στο ένα από τα δύο if. Τοποθετώντας τα στο αναδιατασσόμενο και τρέχοντας τον κώδικα παρατηρήσαμε ότι οι συνολικοί κύκλοι ρολογιού αυξήθηκαν. Αυτό οφείλεται στο γεγονός ότι οι κύκλοι που καταναλώνει κάθε επανάληψη του συγκεκριμένου βρόχου while είναι περίπου 129 και άρα η μετάβαση από και προς το αναδιατασσόμενο του προσθέτει κύκλους (συγκεκριμένα μετά την είσοδο στο αναδιατασσόμενο ο βρόχος while διαρκεί περίπου 218 κύκλους).

Η δεύτερη συνάρτηση την οποία προσπαθήσαμε να βελτιστοποιήσουμε ήταν η order_update(), η οποία ταξινομεί τις μεταβλητές με βάση το activity τους. Η συνάρτηση αυτή αποτελείται από μια σειρά αναθέσεων τιμών με και χωρίς χρήση δεικτών και από ένα βρόχο while μέσα στον οποίο γίνονται αναθέσεις. Έτσι στη συνάρτηση αυτή χρησιμοποιώντας "προσαρμοζόμενες" εντολές τοποθετούμε στο αναδιατασσόμενο τις

αναθέσεις που γίνονται μέσα στο βρόχο while. Η δομή του κώδικα δε μας επιτρέπει να χρησιμοποιήσουμε τα επόμενα δύο βήματα βελτιστοποίησης, δηλαδή την εκτέλεση υπολογισμών σε παραλληλισμό και το manual loop unrolling. Επίσης εφόσον η ανάγνωση των δεδομένων γίνεται από αρχείο δε μπορούμε να χρησιμοποιήσουμε μνήμη υψηλών επιδόσεων και έτσι φτάνουμε απευθείας στο τελευταιό βήμα βελιστοποίησης, δηλαδή στην αλλαγή του optimization level σε O3. Τα αποτελέσματα φαίνονται στον Πίνακα 4.2.

Βήμα Βελτιστοποίησης	Συνολικοί Κύκλοι	Data Cache Misses	Ποσοστό Βελτιστοποίησης(%)
Αρχικά	2593452	74310	--
Χρήση προσαρμοζόμενων εντολών	2593381	74734	0,003
Βελτιστοποίηση του compiler	1572806	75654	39,4

Πίνακας 4.2

Η συνεχής χρήση δεικτών και οι κλήσεις συναρτήσεων μας αποτρέψανε από περαιτέρω βελτιστοποίηση του συγκεκριμένου κώδικα. Έτσι ο κώδικάς μας εκτελείται σε 5,24 msec. Όπως και για την περίπτωση του TSP οι συναρτήσεις order_update() και order_select() δίνονται στο Παράρτημα.

5.3 Συμπεράσματα

Όπως και στην περίπτωση του TSP έτσι και εδώ παρατηρούμε ότι η δομή του κώδικα είναι το μεγαλύτερο εμπόδιο για τη βελτιστοποίηση του κώδικα. Συγκεκριμένα, ενώ καταφέραμε να κάνουμε τον κώδικα να τρέχει περίπου 3 φορές γρηγορότερα από τον αρχικό κώδικα σε C, η συνεχής χρήση δεικτών και οι πολλές κλήσεις συναρτήσεων ήταν οι βασικοί παράγοντες που μας απέτρεψαν από μεγαλύτερες βελτιστοποιήσεις. Θεωρούμε μάλιστα ότι μια αλλαγή στον κώδικα και μετατροπή του από δυναμική σε στατική υλοποίηση και ίσως μείωση ορισμένων από τις συναρτήσεις μπορούν να οδηγήσουν στην αύξηση των τμημάτων του κώδικα που μπορούν να εισαχθούν στο αναδιατασσόμενο κομμάτι και συνεπώς στη μείωση των συνολικών κύκλων ρολογιού. Ένας τελευταίος παράγοντας είναι το γεγονός ότι οι πράξεις (όπως π.χ. διαίρεση και πολλαπλασιασμός) που καταναλώνουν τους περισσότερους κύκλους βρίσκονται διάσπαρτες στον κώδικα πράγμα το οποίο δε μας επιτρέπει να τις τοποθετήσουμε στο αναδιατασσόμενο. Έτσι μια μετατροπή του κώδικα με στόχο τη συγκέντρωση, όσο αυτό είναι δυνατό, των πράξεων σε ένα συγκεκριμένο σημείο του κώδικα μπορεί να οδηγήσει σε μεγαλύτερες βελτιστοποιήσεις εφόσον πλέον το τμήμα αυτό θα μπορεί να εισαχθεί στο αναδιατασσόμενο κομμάτι.

ΚΕΦΑΛΑΙΟ 6

Βελτιστοποίηση Γενετικού Αλγορίθμου

Στο κεφάλαιο αυτό θα παρουσιάσουμε την υλοποίηση και τη βελτιστοποίηση του γενετικού αλγορίθμου με τις μεθόδους που περιγράψαμε στην ενότητα 3.4.

6.1 Υλοποίηση Γενετικού Αλγορίθμου

Πριν ξεκινήσουμε την ανάλυση της υλοποίησης καλό θα ήταν να υπενθυμίσουμε μερικά από τα βασικά στοιχεία που χρησιμοποιήσαμε. Το πρόβλημα το οποίο επιλύουμε με τη χρήση γενετικού αλγορίθμου είναι αυτό του πλανόδιου πωλητή. Για άλλη μια φορά αποφασίσαμε ότι δε θα κάνουμε ανάγνωση των δεδομένων εισόδου από αρχείο, αφού αυτό καταναλώνει πολλούς κύκλους, αλλά από ένα header file, καθώς επίσης και η επιλογή των τιμών για τις οποίες εκτελέσαμε τον κώδικα έγινε με γνώμονα την ταχύτητα. Έτσι τρέξαμε τον κώδικα για 10 πόλεις, με έναν πληθυσμό που αποτελείται από 50 άτομα, δημιουργήσαμε 10 γενιές και 5 επαναλήψεις. Όπως και προηγουμένως έτσι και τώρα θα πούμε λίγα λόγια τα αρχεία τα οποία αποτελούν το πρόγραμμά μας και τι περιέχουν. Το πρόγραμμά μας αποτελείται από τρία αρχεία:

- data.h : το αρχείο με τα δεδομένα εισόδου. Εδώ δίνουμε τιμές στις μεταβλητές:
    - ♦ CIT : περιέχει τον αριθμό των πόλεων για τον οποίο εκτελούμε τον κώδικα.
    - ♦ POP : εδώ καταχωρούμε το μέγεθους του πληθυσμού που θέλουμε να δημιουργήσουμε.
    - ♦ R_SEED : τιμή η οποία χρησιμοποιείται στην αρχικοποίηση της γεννήτριας τυχαίων αριθμών ώστε κάθε φορά να παράγει διαφορετική ακολουθία αριθμών.
    - ♦ GEN : ο αριθμός των γενεών για τον οποίο θέλουμε να εκτελέσουμε τον κώδικα.
    - ♦ RUNS : στη μεταβλητή αυτή καταχωρούμε τον αριθμό των επαναλήψεων του κώδικα.
    - ♦ M_PROP : μεταβλητή που περιέχει την πιθανότητα να έχουμε μετάλλαξη.
    - ♦ M_PERC : περιέχει την επί τοις εκατό (%) πιθανότητα να έχουμε μετάλλαξη για χρήση με τη συνάρτηση myrand() την οποία δημιουργήσαμε εμείς.
    - ♦ map[CIT][CIT] : πίνακας μεγέθους CITxCIT (όπου CIT ο αριθμός των πόλεων που αναφέρθηκε προηγούμενως) ο οποίος περιέχει ουσιαστικά τη βιβλιοθήκη TSPLIB.
- main.h : στο αρχείο αυτό ουσιαστικά βρίσκεται όλη η εφαρμογή. Περιέχει τόσο τη δημιουργία του πληθυσμού (με τη συνάρτηση PopGen()), όσο και τη συνάρτηση Genetic(),με την κλήση της

οποίας έχουμε την έναρξη του γενετικού αλγορίθμου και την επιλογή των δύο καλύτερων και δύο χειρότερων ατόμων του πληθυσμού. Επίσης περιέχει τις συναρτήσεις:

- ✓ WeightCalc() : υπολογίζει το βάρος της κάθε διαδρομής.
- ✓ cross() : χρησιμοποιείται ώστε να γίνει η διασταύρωση των δύο γονέων.
- ✓ mutate() : υλοποιεί τη μετάλλαξη με τη μέθοδο 2-opt.
- ✓ main() : η κύρια συνάρτηση του προγράμματος η οποία καλεί τις PopGen() και Genetic() και επιστρέφει τα αποτελεσμάτα στο τέλος του προγράμματος.

- twoopt.h : στο αρχείο αυτό περιέχεται ο κώδικας της μεθόδου 2-opt.

6.2 Γενετικός Αλγόριθμος και Stretch

Για άλλη μια φορά προκειμένου να βρούμε τα τμήματα του κώδικα που καταναλώνουν τους περισσότερους κύκλους κάναμε το profiling του κώδικα και πήραμε τα αποτελέσματα του σχήματος 5.1.

(Gprof Profile) Show All

%	Cumulative Cycles (K)	Self Cycles (K)	Calls	Self Cycles/call (K)	Total Cycles/call (K)	Name	Graph
43.52	1463.75	1463.75	2700	0.54	0.54	__divdf3	
12.67	1889.75	426.00	350	1.22	1.22	WeightCalc	
10.96	2258.36	368.61	5	73.72	465.16	PopGen	
10.06	2596.64	338.28	2600	0.13	0.13	__muldf3	
5.96	2797.10	200.46	5	40.09	133.68	Genetic	
3.70	2921.65	124.55	50	2.49	9.36	cross	
3.05	3024.34	102.69	2700	0.04	0.04	__muldi3	
2.74	3116.38	92.04	5250	0.02	0.02	__floatsidf	
2.65	3205.41	89.03	10	8.90	8.90	twoopt	
1.61	3259.45	54.04	2700	0.02	0.06	rand	
1.36	3305.02	45.58	2600	0.02	0.02	__fixdfsi	

Σχήμα 5.1

Παρατηρώντας το profiling αυτό βλέπουμε ότι οι περισσότεροι κύκλοι καταναλώνονται σε διαιρέσεις, πράγμα το οποίο ήταν και αναμενόμενο εφόσον ο γενετικός αλγόριθμος χρησιμοποιεί σε πολλά σημεία τη γεννήτρια τυχαίων αριθμών, π.χ. για τη δημιουργία του πληθυσμού, την επιλογή των σημείων τομής στους γονείς για τη διασταύρωση κτλ. Έτσι αρχικά το ενδιαφέρον της προσπάθειάς μας στράφηκε στη βελτιστοποίηση της γεννήτριας τυχαίων αριθμών. Την πρώτη φορά που εκτελέσαμε το πρόγραμμά μας χρησιμοποιήσαμε τη γεννήτρια τυχαίων αριθμών που παρέχουν έτοιμη το Microsoft Visual Studio και το Stretch IDE, όμως για να μπορέσουμε να την τοποθετήσουμε στο αναδιατασσόμενο κομμάτι έπρεπε να υπάρχει σαν κομμάτι μέσα στον κώδικα, οπότε και την αντικαταστήσαμε από τη συνάρτηση myrand() την οποία και δημιουργήσαμε για αυτό το σκοπό. Όπως είναι γνωστό μια γεννήτρια τυχαίων αριθμών αρχικά υπολογίζει μια τιμή seed και στη συνέχεια επιστρέφει το υπόλοιπο της διαίρεσης του seed με κάποιον αριθμό, στο αναδιατασσόμενο κομμάτι όμως οι τελεστές / (διαίρεση) και % (υπόλοιπο,modulo) δεν υποστηρίζονται έτσι η διαίρεση γίνεται με ολίσθηση προς τα δεξιά και ο διαιρέτης πρέπει να είναι δύναμη του 2, όμως στο πρόγραμμά μας ο διαιρέτης δεν είναι πάντα δύναμη του 2. Συνεπώς στο αναδιατασσόμενο προσπαθήσαμε να τοποθετήσουμε τον υπολογισμό των seeds που όμως είναι ένας πολύ απλός κώδικας και έτσι η τοποθέτησή του στο αναδιατασσόμενο αντί να μειώσει τους κύκλους ρολογιού είχε ως αποτέλεσμα να τους αυξήσει.

Μια δεύτερη σκέψη που έγινε με σκοπό τη βελτιστοποίηση του κώδικα ήταν αντί να καλούμε τη συνάρτηση myrand() κάθε φορά που τη χρειαζόμαστε και να δημιουργούμε κάθε φορά το κατάλληλο seed, να

υπολογίσουμε τον αριθμό των seeds που χρειάζεται το πρόγραμμα και να τα παράγουμε όλα μάζι στην αρχή του προγράμματος. Για το συγκεκριμένο πρόγραμμα ο αριθμός των seeds που θέλουμε να δημιουργήσουμε δίνεται από τη σχέση (CIT*POP+4*GEN)*RUNS. Έτσι η συνάρτηση myrand() αντικαταστάθηκε από τη συνάρτηση generateSeeds() και στα σημεία του κώδικα που καλούσαμε τη myrand() πλέον πραγματοποιούμε τον υπολογισμό του υπολοίπου της διαίρεσης (modulo) τις τιμής που επιστρέφει η generateSeeds με τον κατάλληλο αριθμό.

Πλέον το επόμενο βήμα ήταν να τοποθετήσουμε τον κώδικα της generateSeeds() στο αναδιατασσόμενο. Έτσι κάναμε χρήση των "προσαρμοζόμενων" εντολών. Όμως αντί μέσα στο αναδιατασσόμενο να υπολογίζουμε ένα seed κάθε φορά, υπολογίζουμε οκτώ. Αυτό έγινε με σκοπό την ελαχιστοποίηση των μεταβάσεων από και προς το αναδιατασσόμενο, ενώ ταυτόχρονα μας δόθηκε η δυνατότητα να χρησιμοποιήσουμε το δεύτερο βήμα βελτιστοποίησης δηλαδή την εκτέλεση υπολογισμών σε παραλληλισμό.

Το τρίτο βήμα βελτιστοποίησης περιλάμβανε την τοποθέτηση των τιμών της μεταβλητής map στη μνήμη cache του επεξεργαστή (χρήση μνήμης υψηλών επιδόσεων).

Το τέταρτο βήμα βελτιστοποίησης έγινε αλλά με τη χρήση ενός περιορισμού. Δηλαδή υλοποιήσαμε manual loop unrolling και ξεδιπλώσαμε το βρόχο 4 φορές. Αυτό είχε ως αποτέλεσμα την εισαγωγή του περιορισμού (CIT*POP + 4*GEN)*RUNS ≥ 32 γιατί με τον τρόπο

αυτό υπολογιζούμε 32 seeds από την πρώτη εκτέλεση του κώδικα. Ο περιορισμός αυτός δεν είναι απαγορευτικός γιατί ικανοποιείται πάντα.

Τέλος κάναμε χρήση του optimization του compiler. Τα αποτελέσματα που πήραμε παρουσιάζονται στον πίνακα 5.1. Επίσης δίνουμε τους κύκλους που έπαιρνε το πρόγραμμα όταν χρησιμοποιούσαμε τη γεννήτρια τυχαίων αριθμών του Stretch IDE και όταν την αντικαταστήσαμε με τη myrand(). Τα ποσοστά βελτιστοποίησης υπολογίζονται από τη στιγμή που γίνεται χρήση της generateSeeds().

Βήμα Βελτιστοποίησης	Συνολικοί Κύκλοι	Data Cache Misses	Ποσοστό Βελτιστοποίησης(%)
Με rand της Stretch IDE	3363499	3290	--
Με myrand()	2100930	3301	--
Με generateSeeds()	1555753	9321	--
Με χρήση προσαρμοζόμενων εντολών και εκτέλεση εντολών με παραλληλισμό	1532706	9319	1.48
Με χρήση μνήμης υψηλών επιδόσεων	1532536	9084	1.49
Με manual loop unrolling	1530139	9098	1.64
Βελτιστοποίηση του compiler	605821	9066	61.1

Πίνακας 5.1

Στη συνέχεια προσπαθήσαμε να βελτιστοποιήσουμε τις συναρτήσεις WeightCalc() και PopGen(). Η WeightCalc() αποτελείται από ένα βρόχο for μέσα στον οποίο υπάρχει μια συνθήκη if…else και κάποιες αναθέσεις. Τοποθετώντας τη μέσα στο αναδιατασσόμενο παρατηρήσαμε ότι η μετάβαση από και προς το αναδιατασσόμενο έχει ως αποτέλεσμα την αύξηση των κύκλων της συνάρτησης.

Με παρόμοιο τρόπο διαπιστώσαμε ότι και η τοποθέτηση της PopGen() στο αναδιατασσόμενο κομμάτι προκαλούσε αύξηση των κύκλων της συνάρτησης. Αυτό οφείλεται στο γεγονός ότι αποτελείται από ένα βρόχο for ο οποίος περιέχει μέσα του τέσσερις νέους βρόχους for οι οποίοι εκτελούν απλές λειτουργίες, οι περισσότεροι είναι μια γραμμή. Άρα και πάλι η μετάβαση από και προς το αναδιατασσόμενο είχε ως αποτέλεσμα την πρόσθεση κύκλων στους συνολικούς κύκλους της συνάρτησης. Επίσης στη συνάρτηση αυτή γίνεται ο υπολογισμός του υπολοιπου της διαίρεσης των seeds που πήραμε από τη συνάρτηση generateSeeds() με έναν αριθμό j ο οποίος όμως δεν είναι πάντα δύναμη του 2 και αυτό μας απαγορεύει να τοποθετήσουμε την πράξη στο αναδιατασσόμενο κομμάτι.

Τελικά είδαμε ότι ο συνολικός χρόνος εκτέλεσης του προγράμματος μετά τη βελτιστοποίηση είναι 2,02 msec. Οι συναρτήσεις που μελετήσαμε παρουσιάζονται στο παράρτημα.

6.3 Συμπεράσματα

Με βάση τα αποτελέσματα που πήραμε στην ενότητα 6.2 ο κώδικάς μας τρέχει περίπου 6 φορές γρηγορότερα από τον αρχικό κώδικα σε C. Και στην περίπτωση αυτή ο απλός κώδικας με συναρτήσεις οι οποίες δεν περιείχαν σύνθετους υπολογισμούς αλλά αντίθετα απλούς υπολογισμούς οι οποίοι έπρεπε να εκτελεστούν πολλές φορές ήταν ο κύριος λόγος εξαιτίας του οποίου δε μπορέσαμε να τοποθετήσουμε ορισμένα τμήματα του κώδικα στο αναδιατασσόμενο κομμάτι. Ένας άλλος παράγοντας είναι το γεγονός ότι όπως είπαμε στο αναδιατασσόμενο κομμάτι δεν υποστηρίζονται οι τελεστές / και % αλλά για να κάνουμε διαίρεση ουσιαστικά κάνουμε ολίσθηση προς τα δεξιά με έναν αριθμό ο οποίος πρέπει να είναι δύναμη του 2. Στον κώδικά μας όμως ο διαιρέτης μπορεί να είναι οποιοσδήποτε αριθμός και συνεπώς δε μπορούμε να εγγυηθούμε ότι είναι δύναμη του 2. Συνεπώς δε μπορούσαμε να τοποθετήσουμε ένα σημαντικό ποσοστό πράξεων στο αναδιατασσόμενο. Τέλος, πρέπει να πούμε ότι το μεγαλύτερο ποσοστό αυτής της βελτιστοποίησης οφείλεται στις βελτιστοποιήσεις του compiler γεγονός το οποίο εξηγείται γιατί ο compiler ξεδιπλώνει τους βρόχους που περιέχει ο κώδικας.

ΚΕΦΑΛΑΙΟ 7

Συμπεράσματα και Μελλοντική Εργασία

Πριν ολοκληρώσουμε την παρούσα εργασία θα κάνουμε μια ανασκόπηση στα συμπεράσματα που βγάλαμε τόσο όσο αφορά την υλοποίηση μας και τα αποτελέσματά της όσο και για τα εργαλεία της Stretch, καθώς επίσης θα παραθέσουμε και μερικές ιδέες για περαιτέρω επέκταση της συγκεκριμένης εργασίας.

7.1 Συμπεράσματα

Η εργασία αυτή είχε ως στόχο τη μέλετη των αλγορίθμων TSP, MINISAT και γενετικού και το κατά πόσο αυτοί μπορούν να βελτιστοποιηθούν με τα εργαλεία της Stretch.

Κατά τη διάρκεια της εργασίας αυτής δε μπορούμε να πούμε ότι εντοπίσαμε προβλήματα στο γράψιμο του κώδικα που υλοποιεί τους αλγορίθμους TSP και γενετικό (εφόσον χρησιμοποιήσαμε τον πρωτότυπο κώδικα του MINISAT), αλλά προβλήματα αντιμετωπίστηκαν κατά την προσπάθεια βελτιστοποίησης κυριώς λόγω του τρόπου με τον οποίο είχαν υλοποιηθεί οι κώδικες και τον περιορισμών που επέβαλε η ίδια η

Stretch. Για τους δύο αυτούς λόγους παρόλο που καταφέραμε να βελτιστοποιήσουμε τους κώδικες, οι βελτιστοποιήσεις αυτές δεν ήταν πολύ μεγάλες.

Όπως αναφέραμε στα προηγούμενα κεφάλαια οι κύριοι λόγοι που μας εμπόδισαν σε περαιτέρω βελτιστοποιήσεις, εκτός φυσικά από την δομή του κάθε κώδικα, είναι το γεγονός ότι το αναδιατασσόμενο κομμάτι της Stretch είναι περιορισμένου μεγέθους, δεν υποστηρίζει κάποιες πράξεις και έχει περιορισμένο αριθμό καταχωρητών που μπορούν να χρησιμοποιήθουν για είσοδο και έξοδο από αυτό.

Συνεπώς, θα μπορούσαμε να περιμένουμε καλύτερα αποτελέσματα αν η Stretch μας παρείχε κάποιες επιπλέον δυνατότητες όπως:

- Απευθείας πρόσβαση στη μνήμη. Με τον τρόπο αυτό δε θα είχαμε περιορισμένο αριθμό εισόδων στο αναδιατασσόμενο κομμάτι, συγκεκριμένα ως είσοδο μπορούμε να έχουμε μόνο τρείς 128-bit καταχωρητές, και θα μπορούσαμε να εκτελούμε περισσότερες πράξεις εφόσον όλα τα δεδομένα θα ήταν προσπελάσιμα μέσω της μνήμης.
- Μεγαλύτερο αναδιατασσόμενο κομμάτι. Αν ο αριθμός των υπολογιστικών πόρων στο αναδιατασσόμενο είναι μεγαλύτερος θα μπορούσαμε και πάλι να εκτελούμε περισσότερους υπολογισμούς.
- Υποστήριξη περισσότερων αριθμητικών πράξεων. Όπως είδαμε πράξεις όπως η διαίρεση και το υπόλοιπό της δε μπορούν να υλοποιηθούν στο αναδιατασσόμενο αν ο διαιρέτης δεν είναι δύναμη του 2.

- Μεγαλύτερη πρόσβαση στο αναδιατασσόμενο κομμάτι από το χρήστη. Πιο συγκεκριμένα, αν η Stretch παρείχε στο χρήστη τη δυνατότητα να επέμβει στον κώδικα της assembly των περιεχομένων του αναδιατασσόμενου ίσως να ήταν δυνατό να έχουμε μεγαλύτερες βελτιστοποιήσεις.

7.2 Μελλοντική Εργασία

Γενικά, στη συγκεκριμένη εργασία υπάρχουν ορισμένα τμήματα που θα μπορούσαν να αναλυθούν περισσότερο, έτσι προτείνουμε μερικές ιδέες οι οποίες αν εφαρμοστούν ίσως να πάρουμε καλύτερα αποτελέσματα.

- Αρχικά, θα μπορούσαμε να κάνουμε κάποιες αναδιατάξεις στους υπάρχοντες κώδικες, έχοντας υπ' όψιν τους περιορισμούς που εισάγει η Stretch. Πιο συγκεκριμένα, κυρίως στο MINISAT και στο γενετικό αλγόριθμο ίσως να είναι δυνατή η αποφυγή ορισμένων συναρτήσεων και η ενσωμάτωση των πράξεών τους μέσα σε άλλες μειώνοντας έτσι τα jump που γίνονται σε επίπεδο assembly στα διάφορα σημεία του κώδικα.
- Επίσης, θα μπορούσαμε να κάνουμε μετατροπή του κώδικα και από δυναμικό προγραμματισμό να χρησιμοποιήσουμε στατικό. Δηλαδή, να αντικαταστήσουμε τους δείκτες που χρησιμοποιούν ο TSP και ο MINISAT με πίνακες κάτι το οποίο θα μας επιτρέψει να εισάγουμε κάποια από τα δεδομένα μας στο αναδιατασσόμενο κάτι που μας απαγορεύει η χρήση των δεικτών.

- Ακόμη, στο κεφάλαιο 2 έχουμε προτείνει διαφορετικούς εναλλακτικούς αλγορίθμους για την αντιμετώπιση του TSP και αντίστοιχα υπάρχουν και άλλοι τρόποι να υλοποιήσουμε το γενετικό αλγόριθμο, ίσως μια διαφορετική προσέγγιση να φέρει καλύτερα αποτελέσματα.
- Επιπρόσθετα, οι επιλογές του αριθμού των δεδομένων εισόδου έγιναν ώστε να παίρνουμε τα αποτελέσματα όσο το δυνατό γρηγορότερα, καλό θα ήταν να μελετήσουμε τους αλγορίθμους με μεγαλύτερα δεδομένα εισόδου και να δούμε πως θα συμπεριφερθούν οι βελτιστοποιήσεις που κάναμε.
- Τέλος μια ενδιαφέρουσα πρόταση θα ήταν να δοκιμάσουμε να τρέξουμε τους αλγορίθμους σε κάποιο άλλο πρόγραμμα όπως π.χ. το cell ή κάποιο άλλο πολυεπεξεργαστικό σύστημα και να δούμε τη συμπεριφορά τους στους συγκεκριμένους μικροεπεξεργαστές.

Κλείνοντας καλό θα ήταν οι τελευταίες γραμμές αυτής της εργασίας να αφιερωθούν για ένα ευχαριστώ στην εταιρεία Stretch η οποία παρέχει δωρεάν τα εργαλεία της για εκπαιδευτικούς λόγους κάτι που αν δε συνέβαινε δε θα ήταν δυνατή η πραγματοποίηση της παρούσας διπλωματικής εργασίας.

Παράρτημα

Στο παράρτημα αυτό παρουσιάζονται τα τμήματα του κώδικα που βελτιστοποίησαμε καθώς και οι συναρτήσεις που υλοποιήσαμε στο αναδιατασσόμενο κομμάτι. Στο τέλος των συναρτήσεων αυτών μπορούμε να δούμε σε μορφή σχολίων τις εντολές που αντικαταστήσαμε.

<u>TSP με 2-opt</u>

1° for-loop

```
for (i=0; i<n2; i++)
        {
          int Input1[4]={i,n,n1,0};        //orismos metablhths gia thn eisagwgh stoixeiwn
                                            //sto anadiatassomeno
          WRAL128X(&C,Input1,0);   //eisagwgh stoixeiwn sto anadiatassomeno
          twoopt1(C,&D);                   //klhsh sunarthshs tou anadiatassomenou
          WRAS32X(D,&Output,0);    //eksagwgh stoixeiwn

          limit=Output;                    //xrhsh apotelesmatwn
          i2 = ptr[i1];
            j1 = ptr[i2];
}
```

Αναδιατασσόμενο

```
SE_FUNC void twoopt1 (WR H,  WR *I )       //dilwnoume SE_FUNC void th
sunarthsh
                                            // pou bazoume sto anadiatassomeno kommati
{
   se_uint<32> Matrix1 = 0;      //dhlosh mias unsigned metablitis 64 bits
   se_sint<32> Matrix2 = 0;
   se_sint<32> Matrix3 = 0;

   if(integer(H(31,0))==0)
         Matrix1 = (se_uint<32>)H(63,32);
  else
         Matrix1 = (se_uint<32>)H(96,64);

*I = (WR)(Matrix1);  //ana8esh se WR gia thn epistrofh ths eksodou apo to
                     //anadiatassomeno
 // limit = (i == 0) ? n1:n;
}
```

2° for-loop

```
for (j=i+2; j<limit; j++) {

           int Array1[4] = {j1,i1,i2,n};
           int Array2[2] = {Max,ptr[j1]};

           WRAL128X(&A,Array1,0);
           WRBL64X(&B,Array2,0);
           two_opt(A,B,&E);
           WRAS128X(E,Array3,0);

           if (Array3[1]) {
                 s1 = i1;
                 s2 = i2;
                 t1 = j1;
                 t2 = Array3[2];
                 Max = Array3[0];
            }
            j1 = Array3[2];
}
```

Αναδιατασσόμενο

```
SE_FUNC void two_opt (WR C, WR D, WR *G )        //dilwnoume SE_FUNC void
th
                                                //sunartisi pou bazoume
                                           //sto anadiatassomeno kommati
{
   se_sint<32> Matrix1 = 0;     //dhlosh mias unsigned metablitis 64 bits
   se_uint<32> Matrix2 = 0;
   se_uint<32> Matrix3 = 0;

   Matrix1 = w[integer(C(95,64)+C(63,32)*C(127,96))] +
             w[integer(D(63,32)+C(31,0)*C(127,96))] –
             (w[integer(C(31,0)+C(63,32)*C(127,96))] +
             w[integer(D(63,32)+C(95,64)*C(127,96))]);
   Matrix2 = (se_uint<32>)(D(31,0) < Matrix1);
   Matrix3 = (se_sint<32>)D(63,32);

        *G = (WR)(Matrix3,Matrix2,Matrix1);      // ana8esh se wide register gia thn
                                      //epistrofh ths eksodou apo to anadiatassomeno
 /*  j2 = ptr[j1];
   max1 = map[i2][i1]+map[j2][j1]-(map[j1][i1]+map[j2][i2]);
        /*
     A better pair of edges has been found,
          thus save them.
   */
 /* if (max1 > max) */
}
```

MINISAT

order_select()

```
static int  order_select(solver* s, float random_var_freq) // selectvar
{
   int*    heap;
   double* activity;
   int*    orderpos;

   lbool* values = s->assigns;
```

```
// Random decision:
    if (drand(&s->random_seed) < random_var_freq)  //ean tyxaio noumero< ths
                                                     //syxnothtas
    {
           int next = irand(&s->random_seed,s->size); //epilogh neas matablhths an den
          assert(next >= 0 && next < s->size);  // einai ektos oriwn toy plhthoys twn var

          if (values[next] == l_Undef)  //nea matablhth prepei na einai undefined
                                         //gia na epilexthei
          return next;
    }

    // Activity based decision:

    heap     = veci_begin(&s->order);
    activity = s->activity;
    orderpos = s->orderpos;


    while (veci_size(&s->order) > 0) // oso yparxoyn vars sto swro
    {
          int    next  = heap[0];
          int    size  = veci_size(&s->order)-1;
          int    x      = heap[size];  // pairnei teleytaia var sto swro

          veci_resize(&s->order,size); //mikrainei swro kata ena

           orderpos[next] = -1;

            if (size > 0)
          {
                 double act    = activity[x];
                 int    i      = 0;
                 int    child = 1;

                 while (child < size)
                 {
                        if (child+1 < size && activity[heap[child]] <
                                                  activity[heap[child+1]])
                       child++;
```

```
                assert(child < size);

                if (act >= activity[heap[child]])      //an acticity last var
                                                      //magalyterh tote break kai
                                                    //mpainei sth thesh tou i sto swro
                break;

                heap[i]          = heap[child];
               orderpos[heap[i]] = i;
                i               = child;
               child            = 2 * child + 1;
            }
        heap[i]          = x;
        orderpos[heap[i]] = i;
      }
      if (values[next] == l_Undef) // pairnei ayto pou einai prwto sto swro prin thn
                                   //anakatataksh
      return next;
    }

    return var_Undef;      // ean den brhke var returns -1
}
```

order_update()

```
static inline void order_update(solver* s, int v) // update_order  me bash activity
{
    int*    orderpos = s->orderpos;           //index in variable order
    double* activity = s->activity;
    int*    heap     = veci_begin(&s->order); // variable order: heap me bash activity
    int     i        = orderpos[v];      // thesh sto dianysma orderpos pou h timh deixnei
thesh
                                         //sto swro
    int     x        = heap[i];          // timh sto swro twn var
    int     parent   = (i - 1) / 2;

    WRA A,B;   //wide registers gia thn eisodo kai thn eksodo sto anadiatassomeno
               //kommati
    int Array2[4];        //thn eisodo kai thn eksodo sto anadiatassomeno kommati
                          //antistoixa
    assert(s->orderpos[v] != -1);
```

```
  while (i != 0 && activity[x] > activity[heap[parent]])  //oso activity paidou
megalyterh
                                                           //apo patera
                                                          //allagh thesh sto swro
  {
     int Array1[4] = {heap[parent],i,parent,0};

     WRAL128X(&A,Array1,0);
     orderupdate(A,&B);
     WRAS128X(B,Array2,0);

     heap[i]          = Array2[0];   // pateras sth thesh toy paidiou
     orderpos[heap[i]] = Array2[1];            //pateras th swsth thesh sto order pos
     i              = Array2[2];        //deikths ston patera
     parent          = Array2[3];   // neos pateras toy patera
  }

   heap[i]     = x;       // telikh thesh tou x ekei pou stamathse loop
   orderpos[x] = i;

}
```

Αναδιατασσόμενο

```
#include <stretch.h>
#include "solver.h"

SE_FUNC void orderupdate(WR C, WR *D)       //dilwnoume SE_FUNC void th
                           //sunartisi pou bazoume sto anadiatassomeno kommati
{
   se_uint<32> Matrix1 = 0;      //dhlosh mias unsigned metablitis  bits
   se_uint<32> Matrix2 = 0;
   se_uint<32> Matrix3 = 0;
   se_uint<32> Matrix4 = 0;

   Matrix1 = (se_uint<32>)C(31,0);         //heap[i] = heap[parent];
   Matrix2 = (se_uint<32>)C(63,32);        //orderpos[heap[i]] = i;
   Matrix3 = (se_uint<32>)C(95,64);        //i = parent;
   Matrix4 = (se_uint<32>)((C(63,32)-1)>>1);  //parent = (i - 1) / 2;

   *D = (WR)(Matrix1,Matrix2,Matrix3,Matrix4);
}
```

Γενετικός Αλγόριθμος

generateSeeds() το manual loop unrolling δίνεται 2 φορές αντί για τέσσερις.

```
void generateSeeds(int* seeds)
{
  int i;
  WRA A,B;
  WRB C;
  int output1[4];
  int output2[4];

   for(i=0;i<((8-(((CIT*POP+4*GEN)*RUNS)%8))
               +((CIT*POP+4*GEN)*RUNS));i=i+16)
   {
      WRAL32X(&A,&seeds[i],0);
      seeds(A,&B,&C);
      WRAS128X(B,output1,0);
      WRBS128X(C,output2,0);

      seeds[i+1]=output1[3];
      seeds[i+2]=output1[2];
      seeds[i+3]=output1[1];
      seeds[i+4]=output1[0];
      seeds[i+5]=output2[3];
      seeds[i+6]=output2[2];
      seeds[i+7]=output2[1];
      seeds[i+8]=output2[0];

      WRAL32X(&A,&seeds[i+8],0);
      seeds(A,&B,&C);
      WRAS128X(B,output1,0);
      WRBS128X(C,output2,0);

      seeds[i+9]=output1[3];
      seeds[i+10]=output1[2];
      seeds[i+11]=output1[1];
      seeds[i+12]=output1[0];
      seeds[i+13]=output2[3];
      seeds[i+14]=output2[2];
      seeds[i+15]=output2[1];
      seeds[i+16]=output2[0];
}
```

Αναδιατασσόμενο

```
#include <stretch.h>
#include "data.h"

SE_FUNC void seeds (WRA D, WRA *E, WRB *F)  //dilwnoume SE_FUNC void th
                                            //sunartisi pou bazoume
                                            //sto anadiatassomeno kommati
{
        se_sint<32> Matrix1 = 0;      //dhlosh mias signed metablitis 32 bits
         se_sint<32> Matrix2 = 0;
        se_sint<32> Matrix3 = 0;
        se_sint<32> Matrix4 = 0;
        se_sint<32> Matrix5 = 0;
        se_sint<32> Matrix6 = 0;
        se_sint<32> Matrix7 = 0;
        se_sint<32> Matrix8 = 0;

        int i=0;

        while(i<8)
        {
                Matrix1=(integer(D(31,0))*257)+361;// seed =(seed * 257)+361;
                Matrix2=(Matrix1*257)+361;
                Matrix3=(Matrix2*257)+361;
                Matrix4=(Matrix3*257)+361;
                Matrix5=(Matrix4*257)+361;
                Matrix6=(Matrix5*257)+361;
                Matrix7=(Matrix6*257)+361;
                Matrix8=(Matrix7*257)+361;
                i++;
        }


         *E = (WR)(Matrix1,Matrix2,Matrix3,Matrix4);      // ana8esh se wide register
                                  //gia thn epistrofh ths eksodou apo to anadiatassomeno
         *F = (WR)(Matrix5,Matrix6,Matrix7,Matrix8);

}
```

WeightCalc()

```
void WeightCalc (int map[CIT][CIT], int population[POP][CIT+1],int i) {
        //Calculates the weights based on the distances of the cities

        int j,w,c1,c2;

        population[i][CIT]=0;

        for (j=0;j<CIT;j++) //Selects a city and its next
        {
                c1=population[i][j];

                if(j==CIT-1)            //if the first city that selects is the last in the
                                        //array then the second city is the city in the first
                                        //position of the array(exp. array[0])
                        c2=population[i][0];
                else
                        c2=population[i][j+1];
                w=map[c1][c2];
                population[i][CIT]+=w;

        }
}
```

PopGen()

```
void PopGen (int population[POP][CIT+1],int *seeds) {      //Generates a random
                                                            //population
        int i,j,r,s,l;
        int cities[CIT];

        for (s =0;s<POP;s++){
                for (i=0;i<CIT;i++) cities[i]=i;
                        for (i=0;i<CIT;i++) { //for each city in cities
                                j=CIT-i-1;
                                if(j==0)
                                        r=0;
                                else
                                {
                                         r=(unsigned int)seeds[counter]%j;
                                        counter++;
                                 }

                                population[s][i]=cities[r]; //copy it to population

                                for(l = r;l<j;l++)
                                cities[l]=cities[l+1]; //remove it from tcities
                }
        }
}
```

Βιβλιογραφία

Παγκόσμιος Ιστός

[1] http://www. research.att.com/~dsj/chtsp TSP Challenge Website

[6] http://en.wikipedia.org

[8] http://lcm.csa.iisc.ernet.in/dsa/node186.html

[9] http://users.cs.cf.ac.uk/C.L.Mumford/howard/FarthestInsertion.html

[10] http://minisat.se

[15] http://students.uta.edu/bx/bxk7163/tsp/tsp.html

[17] http://www.stretchinc.com

Αναφορές

[2] Hend F. Kendela, M. Ayman Al-Ahmar, El-Sayed M. El Horbaty , A Hybrid Heuristic Algorithm for the Traveling Salesperson Problem.

[3] G.Carpaneto, M.dell'Amico, and P.Toth, Exact solution of large-scale, symmetric traveling salesman problems, ACM Transactions on Mathematical Software, Volume 21, Issue 4, December 1994

[4] Gutin and Punnen, The Traveling Salesman Problem and Its Variations, Kluwer Academic publishers, 2002 pp.

[5] A.B. Kahng, Sherief Reda, Match Twice and Stitch: A New TSP Tour Construction Heuristic, Operations Research Letters 32 (2004) 499-509

[7] David S. Johnson, Lyle A. McGeoch, The Traveling Salesman Problem: A Case Study in Local Optimization, November 20, 1995

[11] D. L. Applegate, R. E. Bixby, V. Chvátal and W. J. Cook (2006). The Traveling Salesman Problem: A Computational Study. Princeton University Press.

[12] E. L. Lawler and Jan Karel Lenstra and A. H. G. Rinnooy Kan and D. B. Shmoys (1985). The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik, Chaff: Engineering an Efficient SAT Solver, 2001

[14] Niklas Eén, Niklas Sörensson, An Extensible SAT-solver

[16] J. P. Marques-Silva and K. A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability , IEEE Transactions on Computers, vol. 48, no. 5

[18] Ευστράτιος Φ. Γεωργόπουλος, Σπυρίδων Δ. Λυκοθανάσης, Εισαγωγή στους Γενετικούς Αλγορίθμους, Πάτρα 1999

[19] Σ. Φωτόπουλος, Γενετικοί Αλγόριθμοι – ΓΑ, Πάτρα 2005

[20] Γ. Χρυσός, Υλοποίηση Αλγορίθμων σε Ενσωματωμένες Αρχιτεκτονικές με Σταθερούς και Αναδιατασσόμενους Πόρους, Χανιά 2007

[21] Α. Δόλλας, Γ. Χρυσός, Εργαστηριακές σημειώσεις για το μάθημα: Ενσωματωμένα Συστήματα Υπολογιστών, Χανιά 2006

[22] Stretch Architecture Reference Part I, II & III, Version 1.1